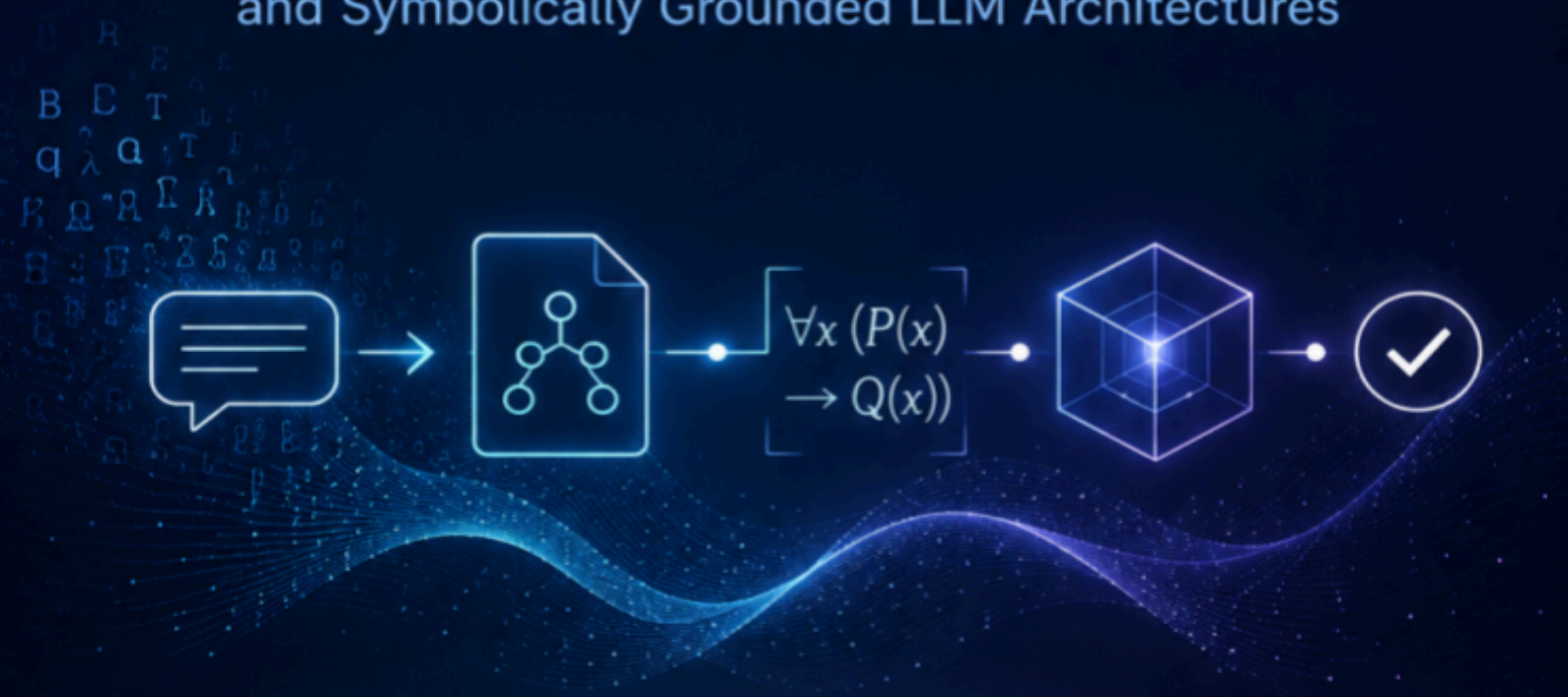
EXECUTABLE NATURAL LANGUAGE

A Critical Survey of Formal Semantics, Abstract Interpretation, Symbolic Execution, and Symbolically Grounded LLM Architectures



EXECUTABLE NATURAL LANGUAGE

A Critical Survey of Formal Semantics, Abstract Interpretation, Symbolic Execution, and Symbolically Grounded LLM Architectures

Toward a Neuro-Symbolic Runtime for Long Documents, Rules, and Agentic Systems

Author's Note

This work was created under the umbrella of Axiologic Research as part of two interconnected research programs, one dedicated to Artificial Intelligence and the other to Outfinitism, both led by Sînică Alboaie. To explore the details of how these two programs intersect and build upon each other, we invite readers to visit the Outfinity Venture Studio page at www.outfinity.ch.

While Artificial Intelligence language models were used to assist with text generation, the foundational philosophy, thematic structure, and analytical approach derive exclusively from the author's decades of dedicated study of social phenomena, social technologies, and genuine hard technologies resulting from various European and self funded research projects. Recently, within the Achilles research project, we have been deeply studying AI generated literature and its automated verification using neuro symbolic approaches; all the free books made available to the public are part of the suite of works we use to test our latest technologies.

Contents

Abstract

Large language models have made natural language an effective interface to computation, yet they have not made natural language executable in the strong sense in which a program, a database query, a transition system, or a logical theory is executable. A model can generate a convincing explanation while changing the scope of a quantifier, omitting an exception, inventing a premise, or applying the same rule differently in two places. External tools improve reliability only when the model has translated the request correctly. A theorem prover can prove the wrong formalization perfectly; a Python interpreter can execute a program that accidentally answers the example while misrepresenting the question; a symbolic executor can expose a real path through software even though the natural-language rule used to construct the path condition was mistranslated. The difficult problem is therefore not merely to add a solver after an LLM. It is to construct an auditable chain from source language to grounded symbols, from grounded symbols to executable formal objects, and from symbolic results back to natural language without allowing the final language model to alter what was established.

This book surveys the relevant traditions as one connected research problem. The earlier foundations include controlled natural languages, model-theoretic and discourse semantics, natural logic, semantic parsing, logic programming, text-to-query systems, program synthesis from examples and language, rule engines, decision tables, temporal logics, automated planning, and neural-symbolic systems. The program-analysis foundations include abstract interpretation, symbolic and concolic execution, dataflow analysis, model checking, SAT/SMT/CSP solving, Datalog and fixpoint evaluation, automata, and proof-producing verification. The recent LLM literature adds program-aided reasoning, declarative solver prompting, autoformalization, first-order-logic translation, temporal-logic translation, formal planning, solver-guided self-repair, LLM-assisted program analysis, LLM-guided symbolic exploration, and the synthesis of verified abstract transformers. Across these bodies of work, the strongest results consistently appear when the target formalism is narrow enough to have an explicit semantics and the accepted output is checked by a component independent of the LLM.

The survey gives special attention to the distinction between two uses of abstract interpretation. The first, already established in NLP, verifies a neural model over a formally defined family of textual perturbations. Interval and relational abstract domains can certify that a classifier’s prediction does not change under every substitution permitted by a specified adversarial set. This is a genuine formal guarantee, but it concerns model stability under a constructed perturbation relation; it does not certify that arbitrary human paraphrases preserve meaning, nor that the model’s original answer is true. The second use is more speculative and more relevant to executable natural language: treating the set of admissible interpretations of a document as a collecting semantics and computing conservative may, must, cannot, and unknown conclusions over that set. This requires explicit domains for identity, time, modality, quantity, source status, normative force, and coverage. It also requires an honest conditional guarantee: the analysis is sound only relative to the interpretation set, ontology, grounding policy, and transfer rules supplied to it.

Symbolic execution is complementary. Abstract interpretation can cheaply show that a violation is impossible under the current abstraction or that a violation remains possible. Symbolic execution can then attempt to construct a concrete interpretation and path that

realizes it. If no such path exists, the failed witness can reveal which abstraction or semantic choice was too coarse. This suggests a research pattern called counterexample-guided semantic refinement: a solver's counterexample, unsatisfiable core, or failed proof is mapped back to the exact source passages and semantic commitments that generated it; the LLM or a human is asked to revise only those commitments; the symbolic analysis is replayed. The LLM is used to search the space of formalizations, while the symbolic kernel determines what follows from each accepted formalization.

Recent systems show important fragments of this vision. PAL delegates procedural computation to Python. SatLM generates declarative constraints for a satisfiability engine. Logic-LM and LINC translate natural-language reasoning tasks into formal logic and rely on symbolic solvers or theorem provers for deduction. NL2TL and nl2spec translate requirements into temporal specifications, with nl2spec explicitly supporting interactive correction at the subformula level. Formal-planning work has shown large gains when LLMs construct planning instances that are then solved and checked by sound tools. SymGPT translates natural-language ERC rules into a restricted grammar and uses symbolic execution to find concrete smart-contract violations, reporting thousands of rule violations and explicit attack paths. In the other direction, AutoBug decomposes program analysis into LLM-handled symbolic paths but remains approximate; SAIL uses LLMs to search for abstract transformers while hard semantic and syntactic checks preserve global soundness. These results are evidence that LLMs are effective proposal and search mechanisms around formal cores, not evidence that probabilistic generation itself has become formal reasoning.

A recent systematic review of neuro-symbolic NLP argues that federative architectures, in which neural and symbolic modules remain autonomous and communicate through structured interfaces, often show larger gains than architectures that merely inject symbolic biases into neural networks. Its authors also caution that the apparent advantage is not yet conclusive because federative and injective systems are usually evaluated on different kinds of tasks. This book adopts the same disciplined position. The architectural case for separation is strong because it clarifies the trust boundary, permits independent replay, and keeps the symbolic component inspectable; the empirical claim that one family universally outperforms another remains unproved [Chatzikyriakidis and Lappin 2026].

The book also provides a critical case study of the public NaturalLanguageLinterAgent, or nllAgent, architecture and its LongTextJS and CircuitJS components. The implementation separates an immutable document-side program from a reusable theory-side program. It preserves exact source anchors, versioned observation types, producer identities, coverage, gaps, and epistemic status. Circuit compilation derives the observations actually demanded by a rule, publication checks whether compatible producers exist, runtime compatibility tests whether the concrete document yielded sufficient observations and coverage, and an independent verifier controls whether a candidate finding is published. Learning occurs in a disposable workspace and cannot publish automatically. These are unusually serious governance decisions for an LLM-assisted document-analysis system. The public repository also records unresolved limitations rather than hiding them. Its current serious-issues register identifies bounded scheduler incrementality, incomplete propagation of interpretation alternatives through every execution path, conservative compatibility analysis for opaque procedural behavior, minimal query planning and temporal indexing, deliberately narrow controlled generation, and mutation support that remains scenario-specific. These are credible research limitations: they concern the semantic and execution substrate, not merely missing user-interface polish.

No independent peer-reviewed evaluation or third-party technical review of nllAgent was located in the searches conducted for this edition. Accordingly, the case-study assessment is explicitly the author's own architectural review, grounded in the public repository, documentation, executable scope, and self-declared serious issues. The judgment is favorable about the separation of evidence, theory, verification, and publication, but cautious about semantic completeness. The strongest current implementation claims concern bounded structural and controlled-English checks. General identity resolution, rich event semantics, current-world knowledge, broad domain transfer, and statistically supported quality claims remain research work rather than established capability.

The book's central proposal is a multi-formalism executable-natural-language runtime. An LLM first proposes a small set of candidate semantic structures, each linked to exact source spans and typed against a selected ontology. A symbolic grounding kernel admits only objects whose identity, provenance, scope, time, modality, and evidence satisfy explicit contracts. A router selects one or more formal targets according to the question: relational or Datalog evaluation for closure and document queries; decision tables for policies; SAT/SMT/CSP for constraints; automata for ordered patterns; model checking for temporal behavior; symbolic execution for path witnesses; and proof assistants or theorem provers for deductive claims. The formal execution produces a canonical result containing accepted facts, rejected candidates, uncertainty, witnesses, proof objects, coverage limits, and assumptions. A final LLM may explain that result, but every sentence in the explanation must be licensed by the canonical structure and traceable to evidence or a verified derivation.

Executable natural language should therefore not be advertised as the execution of unrestricted prose. It should be defined as a controlled compilation, grounding, execution, and realization process over one or more explicit semantic candidates. The guarantee is conditional but useful: given the stated source revision, ontology, grounding, interpretation set, formal semantics, and execution bounds, the reported result follows and can be replayed. The main open research problem is not the solver. It is the construction, comparison, refinement, and validation of the formalizations on which the solver operates.

Preface: Scope, Method, and Critical Position

The ambition of executable natural language is easy to express and easy to overstate. A user writes a rule, policy, scientific hypothesis, procedure, or question in ordinary language; a system understands it; computation follows; the result is returned as ordinary language. Contemporary language models make the beginning and end of this pipeline appear solved. They can read unrestricted prose, generate code and formulas, invoke tools, and explain outputs fluently. The dangerous illusion is that the middle has disappeared. In fact, the middle contains nearly every hard problem: deciding which meaning matters, identifying objects across passages, distinguishing explicit evidence from background assumptions, preserving modality and exceptions, selecting an appropriate formalism, proving that the generated artifact is well formed, and preventing the final explanation from saying more than the execution established.

A useful survey cannot be a catalogue of small systems presented in short isolated subsections. The field is fragmented precisely because the same words are used for different operations. “Symbolic execution” may mean King-style path exploration over symbolic program inputs, the execution of a generated logical form on a knowledge base, or simply the use of an external interpreter. “Formal verification” may refer to the correctness of a solver result, the correctness of a compiled program, the fidelity of natural-language translation, or the robustness of a neural network under a predefined perturbation family. “Grounding” may mean connecting words to database columns, entities to document spans, propositions to world knowledge, or robot instructions to perceptual objects. A serious treatment must show how these layers relate and where guarantees stop.

The survey method therefore follows mechanisms rather than marketing categories. For each line of work, the analysis asks five questions. What artifact is produced from language? What semantics does that artifact have independent of the LLM? How is the artifact grounded in the source or environment? Which component actually determines the result? What is and is not guaranteed when the system succeeds? Papers are treated as evidence for their reported tasks, not as proof of a general architecture. Strong numerical results are included when they clarify the mechanism, but they are not transferred to domains with different semantic structure. The survey horizon is 2 August 2026 and includes peer-reviewed literature, official proceedings, primary research papers, and the public nllAgent repository and documentation.

The term “review” is used carefully. Several broad surveys now exist, but each covers only part of the end-to-end problem. Kuhn’s controlled-natural-language survey organizes languages according to precision, expressiveness, naturalness, and simplicity [Kuhn 2014]. Hamilton and colleagues evaluate whether neuro-symbolic NLP has actually delivered reasoning, transfer, interpretability, data efficiency, and out-of-distribution generalization, while more recent surveys classify task-oriented and data-and-knowledge-driven neuro-symbolic systems [Hamilton et al. 2024; Delvecchio et al. 2025; Wang et al. 2025b]. Surveys of symbolic execution and LLM-assisted program analysis describe the engineering limits of path exploration and the growing role of learned components [Baldoni et al. 2018; Wang et al. 2025a]. A dedicated review studies LLMs as formalizers for automated planning [Tantakoun et al. 2025], and the autoformalization survey concentrates on translating informal mathematics into machine-checkable forms [Weng et al. 2025]. A two-way roadmap for requirements engineering examines both the use of LLMs to make formal methods accessible and the use of formal methods to constrain LLM-produced requirements [Ferrari and Spoletini 2025]. A focused

survey of thirty-five papers maps LLM-assisted formalization of software requirements and specifications, including work targeting Dafny, C, and Java; its scope confirms that translation support is growing faster than end-to-end semantic assurance [Beg et al. 2025]. The 2026 Frontiers systematic review develops a new taxonomy for neuro-symbolic NLP and argues for greater attention to federative systems [Chatzikyriakidis and Lappin 2026].

No surveyed review unifies controlled language, semantic grounding, abstract interpretation, symbolic execution, formalism selection, proof-carrying results, and constrained regeneration as one trust architecture. That conclusion is a literature synthesis rather than a claim of exhaustive nonexistence: terminology differs across fields and relevant papers are distributed across computational linguistics, programming languages, databases, formal methods, software engineering, planning, and robotics. Where no independent review exists, especially for nllAgent, the text labels its judgments as original analysis rather than implying external endorsement.

The book's position is opinionated but conservative. It rejects the idea that a large model should be trusted as the final reasoner merely because its prose resembles a proof. It also rejects the opposite idea that only a hand-written complete semantic parser can support useful formal execution. In open domains, an LLM is often the best available mechanism for proposing events, relations, formal schemas, and candidate mappings. The architectural question is how to constrain that power. The most defensible answer is a federated system with explicit intermediate artifacts, typed and versioned interfaces, independent symbolic execution, conservative unknown states, and replayable evidence.

This position has an important consequence. The system's purpose is not always to produce one definitive formalization. Sometimes the correct artifact is a set of alternatives. A pronoun may refer to two possible actors; a legal phrase may admit competing scopes; a requirement may be incomplete; two sources may disagree; a document may fail to say whether an exception applies. A symbolic runtime that arbitrarily chooses one interpretation is less rigorous than an LLM that candidly states ambiguity. Executability becomes valuable when it can calculate what follows under every admissible reading, what follows under at least one, which conclusions depend on a disputed choice, and what additional evidence would resolve the choice.

The proposed architecture therefore treats uncertainty structurally. Confidence scores may rank candidates, but they do not transform a proposed interpretation into a certified fact. The status of an item records how it became known: directly extracted from a deterministic parser, proposed by a model, assumed for a scenario, derived by a rule, checked by a verifier, or confirmed by a person. Coverage records whether a domain was fully examined, sampled, retrieved, or not processed. A negative conclusion is permitted only when the relevant search is closed under a stated policy. This is the difference between a reliable symbolic system and a fluent classifier with a JSON-shaped output.

The discussion remains accessible rather than proof-heavy. Abstract interpretation, symbolic execution, model checking, and logical semantics are explained through their operational essence and the obligations they create. Formal notation appears only where it clarifies a distinction that prose might blur. The objective is not to present decorative mathematics. It is to give system designers, reviewers, and researchers enough conceptual precision to decide where a proposed guarantee is valid, where it is conditional, and where it is merely aspirational.

The resulting thesis can be stated plainly. LLMs should discover plausible formal shapes; deterministic and formally constrained components should control grounding, compilation, execution, and acceptance; LLMs may then realize the canonical result as language, but only under a contract that prevents unsupported additions. This is not a complete solution to meaning. It is a credible route from probabilistic language processing to bounded, executable, and auditable reasoning.

1. From Fluent Language to Executable Semantics

1.1 What Execution Requires and Where the Trust Boundary Belongs

The word “execute” carries a stronger commitment than “interpret,” “answer,” or “reason.” A program is executable because its operations have a defined effect on a state. A database query is executable because the relational operators determine which tuples are selected, joined, grouped, or excluded. A planning problem is executable because actions have preconditions and effects, and a planner searches for a sequence that reaches a goal. A logical theory is executable in a broader sense because an inference procedure determines whether a conclusion follows, whether a model exists, or whether constraints are satisfiable. In each case, the decisive operation is external to the rhetoric used to describe the task. A result can be reproduced from an artifact and semantics.

An LLM response is different. The model maps a context to a distribution over continuations. It may internalize patterns that resemble algorithms, and it may solve many tasks correctly, but the rule being applied at a particular step is not separately inspectable or guaranteed to be used consistently. Chain-of-thought prompting can improve performance, yet a chain of sentences is not a proof object merely because it is organized as steps. Self-consistency samples several continuations, not several formally defined executions. Tool use can make a value exact while leaving the interpretation that produced the tool call unverified. The crucial distinction is between a model that proposes an executable artifact and a model whose own generation is treated as execution.

Executable natural language therefore begins with materialization. The system must create an object with a declared grammar, type system, semantics, and source map. That object might be a relational query, a logical theory, a state machine, a set of temporal constraints, a decision table, a Datalog program, a finite automaton, or a typed graph with registered operators. The particular choice depends on the task. What matters is that the object’s behavior is no longer determined by the language model’s next-token distribution. It can be rejected, executed, tested, compared, and replayed independently.

This requirement is stronger than ordinary structured output. A model can emit syntactically valid JSON that contains invented entities or mutually inconsistent fields. A schema validates shape, not truth. Similarly, a model can generate legal SQL that joins the wrong columns, valid first-order logic that reverses a quantifier, or a temporal formula that expresses the opposite precedence relation. A useful trust boundary is therefore not simply “after JSON generation.” It lies after source grounding, type checking, semantic compatibility, and formal compilation. Every symbol that affects the answer must have an admissible origin and a defined role.

The architecture naturally separates four authorities. The source authority determines what text, data, or observations were actually supplied. The semantic authority determines which types, relations, modalities, and identity policies are permitted. The rule authority determines which constraints or transformations constitute the theory being executed. The execution authority determines whether a candidate result follows under that theory. An LLM may assist all four, but it should not silently merge them. When the same model invents a premise, selects an

ontology, applies a rule, and writes the verdict, there is no meaningful separation between evidence and conclusion.

Source authority begins with stable bytes and spans. A document analysis system should preserve the exact revision, structure, and location of every cited fragment. It should distinguish a quotation from a summary and a summary from an inference. When a model proposes that a sentence expresses an obligation, the observation must point to the sentence and retain the model, prompt, schema, and alternatives that produced the classification. If the quotation cannot be found in the source, the observation is rejected rather than accepted as a plausible reconstruction. This sounds like mundane provenance engineering, but it is a precondition for every later guarantee.

Semantic authority is usually distributed across an upper ontology and domain packages. A universal ontology is attractive because it promises interoperability, but it tends either to become too abstract to decide useful questions or too large and contentious to govern. A better design defines a small common layer for source spans, mentions, entities, events, time, worlds, status, provenance, alternatives, coverage, and gaps. Domain packages then define concepts such as clinical intervention, regulatory obligation, narrative object event, software API call, or financial transaction. Each package must state not only field names but semantic commitments and tests. A type called “approval” is not meaningful unless the system specifies whether it denotes a document, an act, a state, an authority decision, or some combination.

Rule authority should be versioned separately from document interpretation. A policy, linter theory, or scientific model changes at a different rate from the source it analyzes. If rules are embedded invisibly in prompts, it becomes difficult to know whether two runs applied the same theory. Versioned rule artifacts permit benchmarked publication, semantic diffs, and rollback. They also permit multiple theories over the same document representation. A manuscript can be checked for terminology consistency, causal coherence, and citation support without recompiling it into a different opaque prompt for each rule.

Execution authority should be as small and mechanical as practical. For relational queries, it may be a deterministic evaluator. For constraints, an SMT or CSP solver. For temporal behavior, a model checker. For path-specific program properties, a symbolic executor. For proof obligations, a theorem prover plus a small proof checker. For a custom circuit language, it may be a restricted interpreter with registered operators and independent verifiers. The acceptance component should not ask an LLM whether the result “looks correct.” It should check a witness against the source and theory.

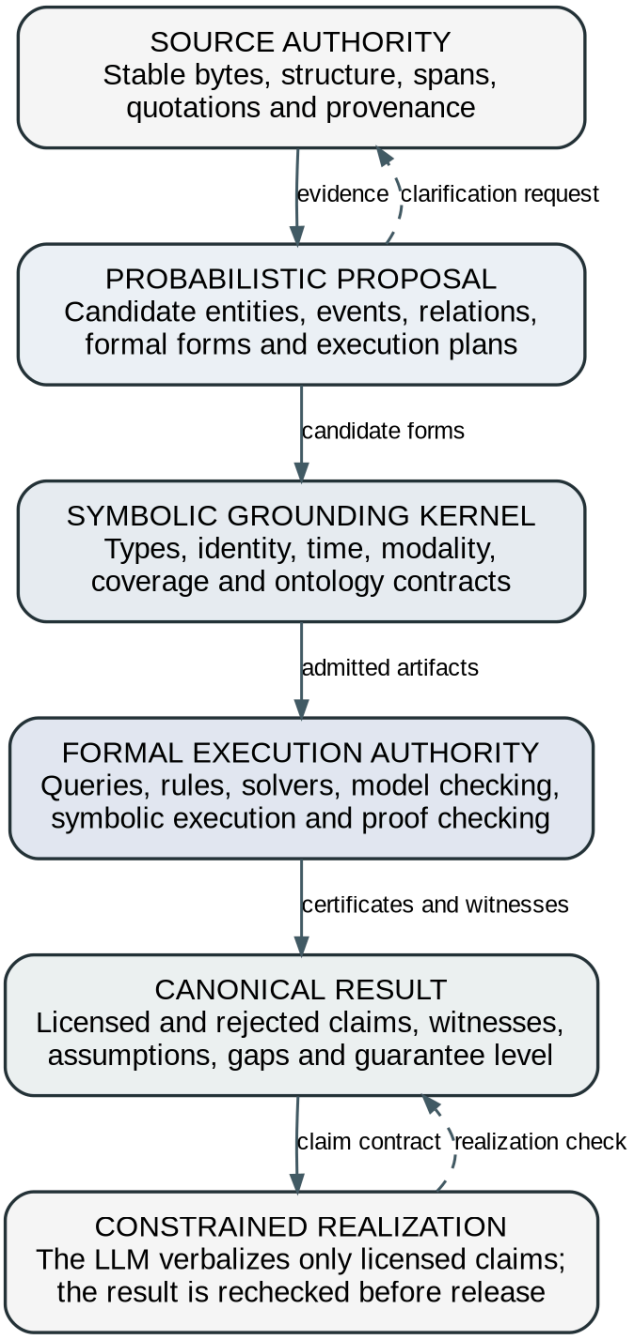


Figure 1. Trust boundary for executable natural language. The LLM proposes semantic forms, while source grounding, formal execution, acceptance, and final claim licensing remain independently checkable.

A useful way to classify systems is by the location of the final oracle. In a purely neural system, the LLM is the oracle. In a tool-augmented system, a tool computes a value, but the LLM may remain the oracle for whether the tool call represents the question. In a solver-backed system, the solver is the oracle for consequences of a generated formalization. In an evidence-grounded system, the solver and grounding kernel jointly determine the accepted result. In a proof-carrying system, even the solver's output can be checked by a smaller trusted kernel. Each step narrows the space in which probabilistic error can silently change the verdict.

This does not imply that every application needs theorem-prover-level assurance. A writing assistant may use lightweight deterministic circuits, while a regulatory or safety system may require formal proof artifacts. The important principle is that the guarantee should match the

mechanism. A deterministic regex check can certify the presence of a literal pattern, not the absence of a concept. A model-assisted event extractor can support a proposed continuity warning, not a mechanically certain account of narrative identity. A solver can establish unsatisfiability of constraints, not completeness of the natural-language translation. Rigor consists largely in refusing to transfer confidence across these boundaries.

The strongest architecture is therefore federative. The neural component is not dissolved into a symbolic network, and the symbolic component is not reduced to a differentiable regularizer. They remain separate systems with a typed interface. The LLM is responsible for breadth, hypothesis generation, semantic normalization, and recovery from unusual language. The formal core is responsible for admissibility, execution, invariants, witnesses, and reproducibility. The 2026 review by Chatzikyriakidis and Lappin calls related systems federative and observes that they often show strong gains on tasks where perception can be separated from inference. The authors also emphasize that comparisons remain confounded by task differences. The architectural argument for federation is nevertheless independent of benchmark superiority: separate modules make claims auditable and enable one component to reject another [Chatzikyriakidis and Lappin 2026].

The final natural-language output requires its own boundary. Once a solver returns a result, it is tempting to provide the trace to an LLM and request a polished answer. Without constraints, the model can add background explanations, infer motives, or simplify uncertainty into certainty. A canonical result should therefore enumerate the claims that may be verbalized, their status, evidence, assumptions, and scope. The renderer may choose wording and organization, but it should be checked against the canonical structure. One practical method is claim-level realization: the model generates sentences associated with result identifiers; a verifier ensures that every sentence is entailed by or paraphrases an allowed claim; unsupported material is removed or labelled as commentary. In high-assurance settings, templated controlled language may be preferable to unrestricted prose.

The trust boundary proposed in this book can be summarized without code. LLMs may propose semantic forms, ontology mappings, formalism choices, repairs, and explanations. They do not establish evidence, close an open-world search, certify identity, decide satisfiability, or authorize a final claim. Those functions belong to explicit symbolic contracts. This separation preserves the unique value of language models while preventing their fluency from being mistaken for proof.

1.2 The Formalization Gap: Why a Correct Solver Can Still Produce the Wrong Answer

The central weakness of most neuro-symbolic reasoning systems is not the inference engine. SAT solvers, SMT solvers, database engines, theorem provers, model checkers, and symbolic executors are often far more reliable than the language model that precedes them. The weakness lies in the formalization gap: the difference between what the source means in context and the formal object submitted for execution. A system can be perfectly sound from formal input to formal output while being unsound from natural language to answer.

Consider a simple policy sentence: “A reviewer may approve a release only after the security report is complete, unless the incident commander authorizes an emergency exception.” A formalizer must preserve at least the actor, action, object, temporal condition, completion state, exception, authority, and scope of “only after.” It must distinguish permission from obligation. It must decide whether the exception removes only the temporal condition or authorizes approval generally. It must connect “the security report” to the correct artifact and “the incident commander” to an identity or role. If any of these choices is wrong, a solver may correctly enforce the wrong policy.

Quantifier scope is a classic source of error. “Every researcher reviewed a paper” can mean that each researcher reviewed at least one paper, not necessarily the same one. “A paper was reviewed by every researcher” has a different structure. Negation creates similar problems: “not every sample passed” does not mean that no sample passed. Conditionals, comparatives, modality, temporal relations, anaphora, and presupposition all create distinctions that surface-level translation can erase. Formal semantic traditions developed sophisticated mechanisms for these phenomena, but open-domain language adds world knowledge, genre conventions, and implicit goals that no single representation captures completely.

The gap also appears in semantic parsing from denotations. A system trained only on a question and final answer may find a program that happens to produce the correct answer on the training environment. Such spurious programs are a known problem in weakly supervised semantic parsing [Pasupat and Liang 2016; Guu et al. 2017; Goldman et al. 2018]. Execution does not distinguish the intended program from an accidental one if both yield the same denotation. Text-to-SQL systems face the same risk: a query may return the right rows because of current data values while containing the wrong join or filter. Execution-guided decoding removes programs that crash or contradict intermediate constraints, but it cannot prove semantic fidelity [Wang et al. 2018].

The formalization gap is best decomposed into separate obligations. Lexical grounding asks which predicates or fields correspond to words and phrases. Referential grounding asks which mentions denote the same entity. Structural grounding asks which arguments, scopes, and dependencies belong together. Temporal grounding connects phrases to instants, intervals, calendars, and ordering. Modal grounding distinguishes facts, possibilities, obligations, permissions, intentions, reported claims, and counterfactuals. Ontological grounding determines whether a concept belongs to the selected domain model. Evidential grounding links every premise to a source span, sensor, database row, or declared assumption. Each obligation can fail independently.

The system must also decide how much meaning to formalize. A total semantic representation of unrestricted prose is neither practical nor necessary for most tasks. The rule

being executed creates demand. A terminology circuit may need only exact spans and normalized terms. A temporal compliance rule may need events, actors, dates, and exceptions. A causal analysis may need claims, interventions, outcomes, and uncertainty. Demand-driven formalization reduces cost and limits the number of unsupported commitments. It also creates an explicit coverage problem: if a rule requires a concept that the available producers cannot materialize, the run must stop or return unknown rather than silently approximate.

Ambiguity should be represented as alternatives, not immediately collapsed. Suppose “Alex told Jordan that they should leave” appears in a procedure. The pronoun may refer to Alex, Jordan, or a group, depending on context and language conventions. A model can rank candidates, but a high confidence is not a proof. If the downstream rule’s verdict is the same under all plausible bindings, the ambiguity is irrelevant and execution can proceed. If the verdict changes, the result should expose the dependency and request clarification or additional context. This is one place where abstract interpretation becomes conceptually useful: it can compute conclusions over a set of interpretations without enumerating every combination in full detail.

A complete mapping is often impossible because natural language is open-world. A document can omit facts that are true, assume common knowledge, or rely on institutional practice. Closed-world reasoning treats unrecorded facts as false; open-world reasoning treats them as unknown. Both policies are appropriate in different contexts. A database inventory may be closed for a specified table and time. A collection of retrieved emails is not generally complete evidence that no notification was sent. The grounding layer must therefore attach coverage to observations and searches. Negative conclusions require a coverage token that states the domain, source revision, scope, method, exclusions, and completeness policy.

External knowledge introduces further complications. A general LLM contains extensive but unversioned and potentially stale information. A symbolic system cannot treat model memory as timeless fact. World knowledge should enter through sourced, dated, jurisdiction-specific, and content-addressed packs. A pack can state that a legal rule was effective in a particular jurisdiction during a particular interval or that a scientific constant was drawn from a named source. Conflicts between packs should remain visible. Fictional or hypothetical worlds must be selectable without contamination by current-world facts. The nllAgent documentation correctly treats such packs as a boundary rather than pretending that a foundation model’s memory provides formal authority.

The fidelity of formalization can be tested, though not universally proved. One method is bidirectional checking: translate the formal object back into controlled language and compare it with the source. Another is contrastive testing: generate minimal textual changes that should change one formal element and preserve the rest. A third is extensional comparison: ask competing formalizations to generate distinguishing examples or query results. A fourth is paraphrase stability: semantically equivalent formulations should compile to equivalent or provably related artifacts. A fifth is counterexample explanation: when the solver produces a witness, map every dependency to source spans and ask whether the scenario is linguistically admissible. A sixth is human review focused on the small set of choices that affect the result rather than the entire formal program.

Interactive systems such as nl2spec illustrate the value of local repair. Instead of asking the user to understand or rewrite an entire temporal formula, the interface associates subformulas with source fragments, allowing a mistaken component to be corrected [Cosler et al. 2023].

Solver feedback can play a similar role. Logic-LM uses syntax and execution errors to prompt re-formalization [Pan et al. 2023]. More advanced systems can use unsatisfiable cores or counterexamples rather than generic error messages. The core insight is that formal feedback should narrow the linguistic question.

The formalization gap also places limits on end-to-end “soundness” claims. A system may prove a theorem of the form: if every accepted observation accurately represents its source and the ontology and rules are correct, then the verdict follows. This is valuable. It should not be rewritten as “the verdict is proven from the document” unless the observation and grounding obligations have also been discharged. The guarantee should name its assumptions and unknowns. A soundness envelope is more honest than a binary trust label.

This book therefore treats formalization as a compilation process with multiple intermediate artifacts rather than a single prompt. The source is segmented and anchored. Candidate observations are produced under schemas. Identity and time alternatives are represented. Ontology compatibility is checked. A target formalism is selected. The compiler produces an executable object and source map. The executor produces a witness, proof, model, or failure. The result is replayed and rendered. At each stage, an error has a specific category and a possible repair. The architecture does not eliminate semantic uncertainty; it prevents uncertainty from being hidden inside fluent output.

The most important research conclusion follows. Solver-backed LLM systems should be evaluated separately on formalization fidelity and formal execution. A high final-answer score can conceal spurious programs. A low score can be caused by a good formalization with an inadequate solver, or by a bad formalization with a perfect solver. Benchmarks need gold or admissible formal structures, evidence alignments, and semantic minimal pairs, not only answers. Until this separation becomes standard, many claims about “formal reasoning with LLMs” will remain difficult to interpret.

2. The Intellectual Lineage of Executable Language

2.1 Controlled Languages, Formal Semantics, Discourse, and Natural Logic

The idea that human-readable language can support mechanical inference predates large language models by decades. Earlier work approached the problem from the opposite direction: rather than asking a statistical model to interpret unrestricted prose, researchers designed languages or semantic theories that limited ambiguity enough to support translation into logic. These traditions are essential because they reveal what an executable-language system must preserve even when an LLM performs the front-end analysis.

Controlled natural languages are the clearest engineering precedent. They restrict vocabulary, grammar, or interpretation while retaining a surface that resembles ordinary language. Attempto Controlled English is the best-known example. Sentences are translated into discourse representation structures and then into formal logic, enabling queries and inference while remaining readable to non-logicians [Fuchs and Schwitter 1996; Fuchs et al. 2008]. Controlled languages have been used for knowledge representation, technical specifications, semantic wikis, business rules, and requirements. Kuhn's survey shows that the family is diverse: some languages are designed for human clarity rather than formal precision, while others trade naturalness for deterministic semantics [Kuhn 2014].

The enduring lesson is not that unrestricted language should be replaced by one controlled dialect. It is that an executable system needs a normalization layer whose semantics is stricter than ordinary prose. An LLM can serve as a translator from open language into a controlled semantic form, but the controlled form must have an independent grammar and interpretation. Without that boundary, "controlled natural language" becomes merely a preferred writing style and provides no computational guarantee.

Controlled languages also expose the importance of author feedback. If a sentence is outside the controlled fragment, a traditional system rejects it or produces an explicit ambiguity. Modern LLM interfaces tend to do the opposite: they select a plausible reading and proceed. Rejection is often treated as a usability defect, yet in a high-assurance system it is a form of epistemic integrity. A next-generation executable-language interface should combine the flexibility of LLM paraphrase with the honesty of a controlled-language compiler. It may propose several normalized readings, explain the distinctions, and ask only when the distinction changes the result.

Formal semantics provides a deeper theoretical lineage. Model-theoretic semantics represents meaning in terms of conditions under which expressions are true in a model. Compositionality connects the interpretation of a complex expression to the meanings of its parts and their syntactic combination. Montague-style semantics, typed lambda calculi, generalized quantifiers, and related frameworks offer precise accounts of scope, argument structure, and logical operators. They demonstrate that natural-language meaning is not reducible to a flat list of triples. Quantifiers, negation, intensional verbs, modality, and context can change the logical behavior of apparently simple phrases.

For executable natural language, formal semantics contributes two requirements. First, the intermediate representation must preserve operator structure rather than only entities and relations. “Every,” “some,” “no,” “only,” “unless,” “may,” “must,” and “before” are not annotations around a proposition; they determine the space of models. Second, semantic construction should be type aware. A temporal interval, a person, an organization, a proposition, and an obligation cannot be freely substituted merely because their surface labels resemble one another. Type checking is one of the cheapest ways to reject incoherent formalizations before invoking a solver.

Discourse representation theory and related dynamic-semantic approaches address phenomena that sentence-by-sentence logic handles poorly. A discourse introduces referents, carries them across sentences, updates context, and creates accessibility constraints for pronouns and descriptions. Boxer and DRS parsers operationalized these ideas for broad-coverage semantic analysis [Bos 2008; Liu et al. 2018]. The Parallel Meaning Bank and shared DRS parsing tasks made it possible to evaluate scoped meaning representations across languages [Abzianidze et al. 2019]. These efforts show both progress and difficulty: variable binding, sense assignment, scope, and discourse relations are substantially harder than shallow extraction.

A document-level executable system should learn from dynamic semantics without requiring one monolithic DRS for every source. It needs persistent discourse objects, scoped entity hypotheses, events, worlds, and update relations. It must distinguish a mention from an entity and a candidate identity link from an established one. It must preserve quoted, hypothetical, negated, and fictional contexts. A flat knowledge graph that merges equal strings into one node will make systematic errors in long documents. Names repeat, aliases occur, roles change, and documents describe mutually inconsistent worlds.

The practical cost of full formal semantic parsing explains why many systems moved toward shallower representations. Semantic role labelling identifies predicate-argument structure. Abstract meaning representation represents concepts and relations in a graph. Information extraction identifies entities, events, and relations. These representations are useful, but their semantics is often incomplete for execution. They may omit quantifier scope, modality, temporal anchoring, and the difference between asserted and reported content. An executable-language runtime should treat such structures as evidence-bearing observations rather than as a complete logical form. A graph can support many local rules while leaving unresolved aspects explicit.

Natural logic offers another important compromise. Instead of translating sentences into an unrestricted formal language, natural-logic systems reason through structured transformations near the surface form. MacCartney and Manning developed inference relations based on monotonicity, lexical entailment, and projectivity [MacCartney and Manning 2007, 2009]. NaturalLI later used natural-logic operators for efficient inference over text [Angeli and Manning 2014]. LangPro combines a formal grammar, lexical knowledge, and theorem proving for natural-language inference [Abzianidze 2017].

Natural logic is attractive because many document questions concern entailment relations that can be captured without constructing a large world model. Replacing “dog” with “animal” is valid in an upward-entailing context but not necessarily under negation or universal restrictions. These calculations are more transparent than an embedding similarity score. They can support contradiction checks, taxonomic reasoning, and controlled inference. However, natural logic is

not a universal executor. It handles certain monotonicity and lexical relations well but does not naturally express arbitrary procedures, arithmetic, planning, or complex temporal behavior. In a multi-formalism architecture, it should be one engine among several.

The semantic traditions also clarify why “grounding” has several levels. Formal semantics can specify how an expression contributes to truth conditions, but it does not automatically determine which real-world object a name denotes or whether a source is credible. DRS can represent a discourse referent, but identity resolution remains an empirical task. Natural logic can propagate lexical entailment, but the lexical relation may be domain dependent. Executability therefore requires a partnership between linguistic structure, domain ontology, and evidence management.

A useful architecture can distinguish at least three semantic products. The first is a source model containing exact structure, spans, and quotations. The second is an interpretation layer containing typed mentions, events, relations, modalities, and alternatives. The third is an executable theory that uses selected interpretations to answer a particular question. Formal semantic techniques primarily inform the second layer. Controlled languages and theorem provers connect the second and third. Provenance and coverage connect both back to the first.

The earlier literature also warns against assuming that one representation can serve every task. DRS is expressive for discourse but expensive to produce and reason over. First-order logic is well understood but awkward for temporal, probabilistic, or defeasible phenomena. Description logics offer decidable ontology reasoning but restrict expressiveness. Natural logic remains close to language but is specialized. Controlled English improves predictability by restricting input. The correct conclusion is not to choose a winner but to preserve enough semantic information to lower a task into the formalism that fits it.

LLMs change the economics of this design. They can generate candidate parses, rewrite uncontrolled language into controlled forms, suggest ontology mappings, and explain formal alternatives to users. They make it practical to attempt semantic interpretation on documents that hand-written grammars would reject. They do not remove the need for the semantic distinctions developed by earlier work. On the contrary, those distinctions become the contracts by which LLM proposals are judged.

A recurring mistake in contemporary systems is to treat an LLM-generated triple store as a complete semantic representation. Triples are convenient for retrieval and graph queries, but they do not by themselves encode scope, provenance, modality, time, negation, identity alternatives, or coverage. An executable semantic layer should be richer but still finite and operational. It can store an assertion as a typed object with participants, temporal context, polarity, modality, world, evidence, producer, and status. It can represent alternative bindings without choosing one. It can declare which dimensions are absent. This is less elegant than a single logical form, but it is more faithful to how real systems acquire meaning.

The deepest contribution of the pre-LLM lineage is therefore conceptual discipline. Human readability does not imply semantic determinacy. Formal determinacy is achieved by restriction, typing, explicit context, and rejection of unsupported constructions. Document meaning is dynamic and scoped. Inference near surface language is possible but specialized. Multiple formal representations are legitimate because they answer different questions. These lessons define the architecture that an LLM-based compiler must recover rather than bypass.

2.2 Semantic Parsing, Executable Queries, Program Induction, and Early Neuro-Symbolic Systems

Semantic parsing converts natural-language utterances into formal objects that can be executed against a database, knowledge base, environment, or interpreter. It is the closest historical predecessor to the executable-language pipeline proposed here. The field includes question answering over tables and knowledge graphs, text-to-SQL, instruction following, regular-expression synthesis, robotic command interpretation, and program generation. Its successes demonstrate the power of external execution; its failures reveal why execution alone does not establish understanding.

Early semantic parsers relied on grammars, lexicons, and supervised logical forms. They could produce transparent representations but required expensive annotations and often transferred poorly across domains. Weak supervision reduced the annotation burden by training from denotations: a question is paired with an answer, and the system searches for a program that yields it. This approach enabled broader datasets, but it created the spurious-program problem. Many programs can accidentally return the same answer, especially in small tables or knowledge bases. The training signal then rewards behavior without identifying the intended semantics [Pasupat and Liang 2016].

The problem is structural, not merely statistical. Suppose the answer to “Which city hosted the 2012 event?” is London. A program selecting the row with year 2012 is intended. A program selecting the first row may also return London in the training table. More data can reduce accidental correlations but cannot guarantee that the learned program reflects the language. Abstract examples, alignment models, type constraints, and execution-based filtering have been developed to suppress spurious candidates [Goldman et al. 2018; Lee et al. 2023]. These techniques are important for LLM formalization as well. A solver result should be treated as one test of a formalization, not as its semantic certificate.

Neural Symbolic Machines combine neural program generation with a symbolic Lisp interpreter [Liang et al. 2017]. The neural component proposes programs, while the executor rejects invalid candidates and returns denotations. Mou and colleagues explicitly coupled distributed neural execution with symbolic execution for natural-language queries, reporting improvements in accuracy, efficiency, and interpretability [Mou et al. 2017]. These systems established a durable pattern: neural models handle language variability; symbolic interpreters enforce operational meaning.

Text-to-SQL made the pattern commercially important. A user question is compiled into a query whose execution returns data. Execution-guided decoding can run partial or complete SQL candidates and discard those that produce errors or impossible results [Wang et al. 2018]. Schema linking constrains names and relations. Modern LLMs substantially improve generation, but cross-domain generalization, ambiguous questions, database-specific conventions, and unanswerable requests remain difficult. The key lesson for executable natural language is that the formal environment itself supplies useful constraints. Column types, foreign keys, and query errors provide feedback that pure language modelling lacks.

Yet text-to-SQL also shows the limits of a single final accuracy metric. Exact string match can reject equivalent queries. Execution accuracy can accept semantically wrong queries that happen to return the same result. A rigorous evaluation needs equivalence under varied database instances or carefully constructed counterexamples. This is directly analogous to

evaluating rule formalization: a generated formal program should be tested on distinguishing scenarios, not only on one observed answer.

Program synthesis from natural language generalizes the idea. Systems generate regular expressions, code snippets, data transformations, or small domain-specific programs. Examples can constrain the candidate space, and type systems can eliminate nonsensical programs. Neural generation of regular expressions showed that language-to-program mapping can work even with limited domain knowledge, while also illustrating brittleness when descriptions are underspecified [Locascio et al. 2018]. Program-aided LLMs later moved computation outside the model, but the fundamental synthesis problem remained: the model must choose a program whose semantics matches the request.

A robust compiler should therefore seek multiple forms of evidence. Natural-language descriptions constrain intent. Input-output examples constrain extensional behavior. Types constrain admissible operations. Domain schemas constrain vocabulary. Tests probe edge cases. A generated program that satisfies all these constraints is more credible than one that merely executes. This multi-evidence principle can be applied to document rules: textual authority, positive and negative examples, expected findings, and semantic mutations should all shape a circuit before publication.

Logic programming provides another executable target. Datalog rules compute consequences to a fixpoint and support incremental evaluation. Prolog and related languages represent relations and search. Abstract interpretation has long been used to analyze logic programs, showing a conceptual bridge between declarative semantics and program analysis. For natural-language applications, logic programming is well suited to monotone closures: transitive relations, taxonomies, dependency propagation, and policy rules without complex defaults. It is less suitable when reasoning depends heavily on uncertainty, inconsistent sources, or nonmonotonic exceptions unless extended with explicit mechanisms.

Decision tables and production-rule systems are often overlooked in academic surveys but are highly relevant. Many policies are not deep logical theories; they are finite combinations of conditions and outcomes. A decision table makes coverage, conflicts, and missing combinations visible. It can be compiled into executable predicates and tested systematically. Natural language can be used to author or explain the table, while the table remains the authoritative semantics. For enterprise document linting, decision tables may be safer and easier to validate than arbitrary first-order logic.

Temporal and procedural instructions require richer targets. Finite-state machines, behavior trees, planning languages, and temporal logics represent ordered actions and conditions. Robotics has long used language parsers that compose executable primitives. Recent neuro-symbolic manipulation systems continue this pattern: a language parser maps an instruction to a program whose components may include neural perception and symbolic counting, spatial reasoning, or action control. The success of such systems depends on a restricted skill library and grounded environment, not on unrestricted semantic coverage [Tziafas and Kasaei 2026].

Probabilistic logic programming and differentiable neuro-symbolic systems attempt tighter integration. DeepProbLog, Logic Tensor Networks, and related systems combine learned predicates with logical structure [Manhaeve et al. 2018; Badreddine et al. 2022]. They are valuable when uncertainty and learning are intrinsic, and they can inject constraints into training. However, differentiability is not equivalent to formal assurance. Fuzzy truth values and soft

penalties can guide models without producing discrete proof obligations. For executable natural language, such systems may estimate or rank semantic candidates, but a high-assurance acceptance boundary still benefits from a separate symbolic checker.

The 2026 neuro-symbolic NLP review distinguishes injective systems, which compile symbolic information into neural models, from federative systems, which preserve autonomous components. It reports that injective systems dominate the literature but often show modest gains on broad NLU tasks, whereas federative systems show larger gains on reasoning tasks. The authors explicitly warn that the comparison is confounded by different benchmarks and that out-of-domain generalization remains a serious problem [Chatzikyriakidis and Lappin 2026]. The balanced interpretation is that federation is architecturally attractive for auditability and task decomposition, but not yet empirically proven as a universal scaling strategy.

The semantic-parsing lineage also clarifies the role of an intermediate language. A representation must be expressive enough to capture task-relevant meaning but constrained enough to compile reliably. If it is too close to surface text, execution remains implicit. If it is too general, translation and solver complexity become unmanageable. Beiser and Penz describe this as an intermediate-language problem in contemporary neuro-symbolic systems: choosing a formalism can move rather than solve the difficulty [Beiser and Penz 2025]. A multi-formalism architecture reduces the pressure on any one language but introduces a new requirement for typed routing and cross-formalism consistency.

A useful semantic pivot should therefore not pretend to be the final logic. It should preserve evidence and task-neutral distinctions—entities, mentions, events, relations, time, modality, quantities, worlds, provenance, alternatives, and coverage—while allowing lowerings into specialized executors. Some queries can be answered directly over the pivot using relational operations. Others require a compiled temporal formula, constraint system, or state machine. The pivot is valuable because it separates language understanding from any particular solver while retaining enough information for source-grounded execution.

The pre-LLM research programme achieved substantial but domain-bounded success. It showed that natural language can be compiled into executable programs, that symbolic interpreters improve systematicity, that types and schemas reduce errors, and that weak supervision creates semantic shortcuts. It also showed that broad coverage and exact semantics are in tension. LLMs substantially improve coverage and candidate generation, but they inherit every structural problem identified by semantic parsing. The correct synthesis is not to discard the earlier machinery. It is to place LLMs in front of and around it, with formal interfaces that make old lessons enforceable at new scale.

3. Abstract Interpretation and Symbolic Execution for Language

3.1 Abstract Interpretation as a Semantics of Bounded Uncertainty

Abstract interpretation was developed as a general theory for computing safe approximations of program behavior [Cousot and Cousot 1977]. A program may have infinitely many inputs and states, making exhaustive execution impossible. Instead of enumerating every state, an analysis represents sets of states through abstract properties such as intervals, signs, possible object types, reaching definitions, or alias relations. Abstract operations are designed so that every concrete behavior relevant to the property remains represented. The analysis may include impossible behaviors and therefore report false alarms, but a sound over-approximation does not silently omit a behavior that the concrete semantics permits.

The operational idea is simple. A concrete execution tracks exact values. An interval analysis might track that a variable lies between zero and ten. When the program adds five, the abstract transfer operation produces an interval between five and fifteen. At a branch, states are refined according to the condition. At a merge, information is joined. For loops and recursive processes, the analysis searches for a fixpoint. Widening can force convergence by sacrificing precision, while narrowing can recover some detail afterward. The mathematical framework supplies a disciplined vocabulary for precision, soundness, convergence, and refinement.

Applying the framework to language requires a concrete semantics. The object being approximated cannot be “meaning” in an undefined sense. For a bounded task, the concrete domain can be the set of semantic structures admissible under a specified grammar, ontology, grounding policy, source revision, and background theory. Each structure contains possible entities, event assignments, temporal relations, modalities, and source links. Ambiguity creates several structures. Missing information leaves dimensions unconstrained. Conflicting sources may generate alternative worlds rather than one inconsistent union.

An abstract semantic state summarizes that set. It may record that a pronoun can refer to either of two people, that an event occurs within a time interval, that an obligation is present under one interpretation and absent under another, or that a quantity lies within a range. A must fact holds in every represented interpretation. A may fact holds in at least one. A cannot fact holds in none. Unknown indicates that the system lacks the representation, coverage, or computational resources needed to decide. This four-way distinction is more informative than a probability alone and more honest than forcing every question into true or false.

Consider a policy document stating: “The operator sends the notice after validation. If emergency mode is active, the supervisor may waive the waiting period.” Suppose it is unclear whether “the waiting period” refers to the period before validation or a later period. A single parse makes a brittle choice. An abstract state can retain two temporal models. If both imply that notice must follow validation, that conclusion is must. If only one permits notice immediately, immediate sending is may. A compliance rule can identify which verdict depends on the ambiguous phrase and request clarification only when necessary.

The primary benefit is not that abstract interpretation magically understands language. It is that it gives a principled execution semantics once a bounded set of readings has been constructed. It tells the system how to aggregate alternatives without confusing possibility with

necessity. It also supports modular domains. Identity, time, quantity, modality, provenance, and normative status can each have their own abstraction, then be combined through reduced products that exchange information where useful.

An identity domain might represent candidate equivalence classes with explicit same, different, and unresolved relations. It must not merge entities merely because names match. A temporal domain might use partial orders, intervals, or difference constraints. A modality domain might distinguish asserted, reported, intended, permitted, required, prohibited, hypothetical, and counterfactual content. A quantity domain might use exact values, intervals, units, and dimensional compatibility. A provenance domain might order evidence statuses such as extracted, proposed, assumed, derived, verified, and human-confirmed without allowing confidence to erase their origin. A coverage domain might track whether a search is complete enough to support absence claims.

The domains should be chosen by the property being checked. A terminology linter does not need a rich deontic lattice. A temporal rule does not need full lexical semantics if events and dates are already materialized. This task-driven selection reduces complexity and mirrors demand-driven static analysis. In a long-document runtime, circuits or rules declare the observation types and abstract dimensions they require. The compiler materializes only those dimensions and rejects the run when a critical domain cannot be supplied.

Transfer functions are the most difficult part. In ordinary program analysis, the semantics of an assignment or arithmetic operation is fixed by the programming language. In natural-language analysis, the operation that updates semantic state may be proposed by an LLM. A sentence could introduce an event, modify a previous state, establish a temporal relation, create an exception, or report someone else's belief. If a learned translator directly becomes the transfer function, classical soundness is lost.

A defensible design separates candidate generation from admitted transfer. The LLM proposes a finite set of typed semantic updates with source spans and alternatives. A deterministic materializer validates that the quoted evidence exists, the fields satisfy a schema, units and types are compatible, references point to admissible objects, and no status is silently upgraded. The abstract interpreter then applies registered transfer functions to the admitted records. The resulting analysis is sound relative to the admitted candidate set, not necessarily relative to every interpretation a human could recognize. This conditionality must be stated explicitly.

There are several ways to strengthen the candidate-set assumption. Multiple independent prompts or models can propose alternatives. Grammar- or ontology-based generators can ensure coverage for controlled fragments. Paraphrase variants can test stability. Contrastive examples can reveal missing interpretations. Human review can focus on result-relevant ambiguities. A compiler can deliberately overgenerate within a bounded grammar and let later constraints eliminate impossible candidates. Each method trades cost for confidence that the set contains all materially plausible readings.

Abstract interpretation is also relevant to document-scale rule propagation. Many linting and compliance problems resemble dataflow analysis. Facts are generated at source locations, propagated through scopes, joined across sections, and invalidated by later statements. A definition may be available after its declaration but not before. An object may change state across a narrative. A requirement may apply within one jurisdiction or version. A claim may

acquire evidence later. An incremental fixpoint engine can compute consequences and update only affected regions when the document changes.

This perspective avoids rerunning a general LLM over an entire document after every edit. Stable structural and semantic artifacts can be cached by content digest. A changed paragraph invalidates local observations and the downstream abstract states that depend on them. The analysis graph identifies which circuits need replay. This is analogous to incremental compilation and static-analysis frameworks. It also makes explanations precise: the system can show which source change altered which abstract invariant and why.

Widening and loss of precision have semantic analogues. A long narrative may contain hundreds of possible object identities or temporal relations. Retaining every combination is impractical. The system may summarize “one of these objects,” “some time within this interval,” or “one of these jurisdictions.” Such widening can introduce spurious warnings. Narrowing or local refinement can recover precision around a suspected violation. The design objective is not to avoid abstraction but to make its precision loss visible and reversible.

The established NLP literature on certified robustness uses abstract-interpretation ideas in a different way. Jia and colleagues used interval bound propagation to train models whose predictions are certified against every word substitution in a predefined family, reporting 75 percent adversarial accuracy on IMDB and SNLI under their threat model [Jia et al. 2019]. Later work used randomized smoothing, robustness-aware embeddings, and specialized Transformer domains to improve certified bounds [Ye et al. 2020; Wang et al. 2023; Huang et al. 2026]. These are genuine guarantees, but the concrete domain is a set of perturbed model inputs, not a set of semantic interpretations. The transformation family is defined externally and assumed to preserve the label.

That distinction matters. Certifying that a classifier’s prediction is invariant under all substitutions in a synonym list does not prove that every substitution preserves the sentence’s meaning in context. It does not prove that the classifier is correct on the original sentence. It certifies robustness relative to a formal perturbation set. This literature is still highly relevant because it demonstrates how abstract domains can scale over combinatorial language inputs and how verification precision depends on representing correlations. It also provides a methodological warning: the strongest formal claim is always relative to the concrete set actually defined.

A promising research direction is to connect the two uses. Certified neural analysis could bound the behavior of the semantic proposal model under controlled paraphrases, while semantic abstract interpretation could bound the consequences of alternative grounded interpretations. The former asks whether a model’s extraction changes under permitted surface transformations. The latter asks whether a document-level conclusion changes across admissible semantic structures. Together they could identify whether uncertainty originates in model instability or genuine semantic ambiguity.

LLMs can also help construct abstract interpreters, but only under mechanical acceptance. SAIL demonstrates this principle by using LLMs to search for abstract transformers while enforcing syntactic and semantic constraints and optimizing a cost that measures unsoundness. The accepted transformers are globally sound, and the system can synthesize precise transformers for nonlinear operators that were not previously available [Gu et al. 2026]. The broader lesson is architectural: generative models can explore an immense design space if a small formal validator controls admission.

Directly asking an LLM to perform abstract interpretation is less reliable. Experiments on program-analysis benchmarks show that models can follow abstract-interpretation styles but make logical errors on complex structures and hallucinate states [Mitchell et al. 2025]. This result should temper proposals in which a model narrates an invariant and the narration is accepted as analysis. The model is valuable for suggesting invariants, domain partitions, or refinement predicates. The fixpoint computation and soundness checks should remain symbolic.

For executable natural language, abstract interpretation is therefore neither a metaphor nor a complete solution. It is a theory for disciplined approximation after the system has defined what counts as a possible interpretation. It supports may/must semantics, incremental propagation, domain composition, conservative absence reasoning, and targeted refinement. Its guarantee is conditional on semantic coverage. This limitation is not fatal; it is the exact point at which model-assisted interpretation must be evaluated and governed.

3.2 Symbolic Execution, Constraint Solving, Model Checking, and Semantic Refinement

Classical symbolic execution executes a program with symbolic inputs rather than specific values [King 1976]. A symbolic state contains a control location, symbolic expressions for program variables, and a path condition describing which inputs reach that state. At a branch, execution forks. One path adds the branch condition; the other adds its negation. A constraint solver determines whether the conditions are satisfiable and can produce concrete inputs that realize a feasible path. The method is powerful because it produces witnesses: not merely a warning that an error may exist, but an input and path that demonstrate it.

The difficulty is path explosion. Branches multiply, loops and recursion create unbounded paths, complex theories challenge solvers, and external libraries are difficult to model. Surveys of symbolic execution describe a large design space of search strategies, state merging, compositional summaries, concolic execution, environment models, and solver optimizations [Baldoni et al. 2018]. No single engine explores every behavior of real software. Symbolic-execution claims therefore include bounds, modelling assumptions, and coverage limitations.

Natural language connects to symbolic execution in several distinct ways. The simplest is language-to-code: an LLM generates a program, and a conventional symbolic executor analyzes that program. Another is language-to-specification: the model generates a property or constraint, which is checked against existing code. A third is semantic-program execution: a document is compiled into a state machine or rule program whose alternative interpretations become symbolic choices. A fourth is LLM-assisted exploration: a model prioritizes paths, proposes summaries, or reasons about constraints that conventional solvers cannot easily express.

The terminology must remain precise. Executing SQL, Python, or a logical form generated from a question is formal execution, but it is not necessarily symbolic execution in the path-sensitive sense. The difference matters because symbolic execution offers path conditions and concrete witnesses, whereas ordinary execution provides one result for one concrete state. Both are useful. A survey that groups them together without distinction makes guarantees appear stronger than they are.

SymGPT provides one of the clearest language-to-specification examples. The system studies 132 natural-language rules from three ERC standards, translates them with an LLM into a restricted grammar, synthesizes constraints that describe violations, and uses symbolic execution to search Ethereum contracts. The authors report 5,783 violations across 4,000 contracts, including 1,375 with explicit attack paths for theft, and report stronger performance than six automated techniques and a security-expert auditing service [Xia et al. 2026]. The strength of the design lies in the narrow grammar, systematic constraint synthesis, and concrete program paths. Its remaining semantic risk lies in whether each ERC rule was translated faithfully.

This pattern generalizes. A policy can be compiled into preconditions, prohibited transitions, and exceptions. A procedure can be compiled into states and actions. A narrative continuity rule can model objects and events, then search for a path in which an object is used after being left behind without retrieval. A clinical protocol can model eligibility and intervention sequences. A

scientific workflow can model data transformations and provenance requirements. In each case, the symbolic executor should operate on a typed program rather than on raw prose.

Symbolic execution is especially useful for counterexamples. Abstract analysis may indicate that a violation is possible because it has lost a correlation. A path search can attempt to choose concrete identities, times, conditions, and exceptions that realize the violation. If the path is satisfiable, the witness can be rendered as a source-grounded scenario. If it is unsatisfiable, the system learns that the abstract warning was spurious under the current semantics.

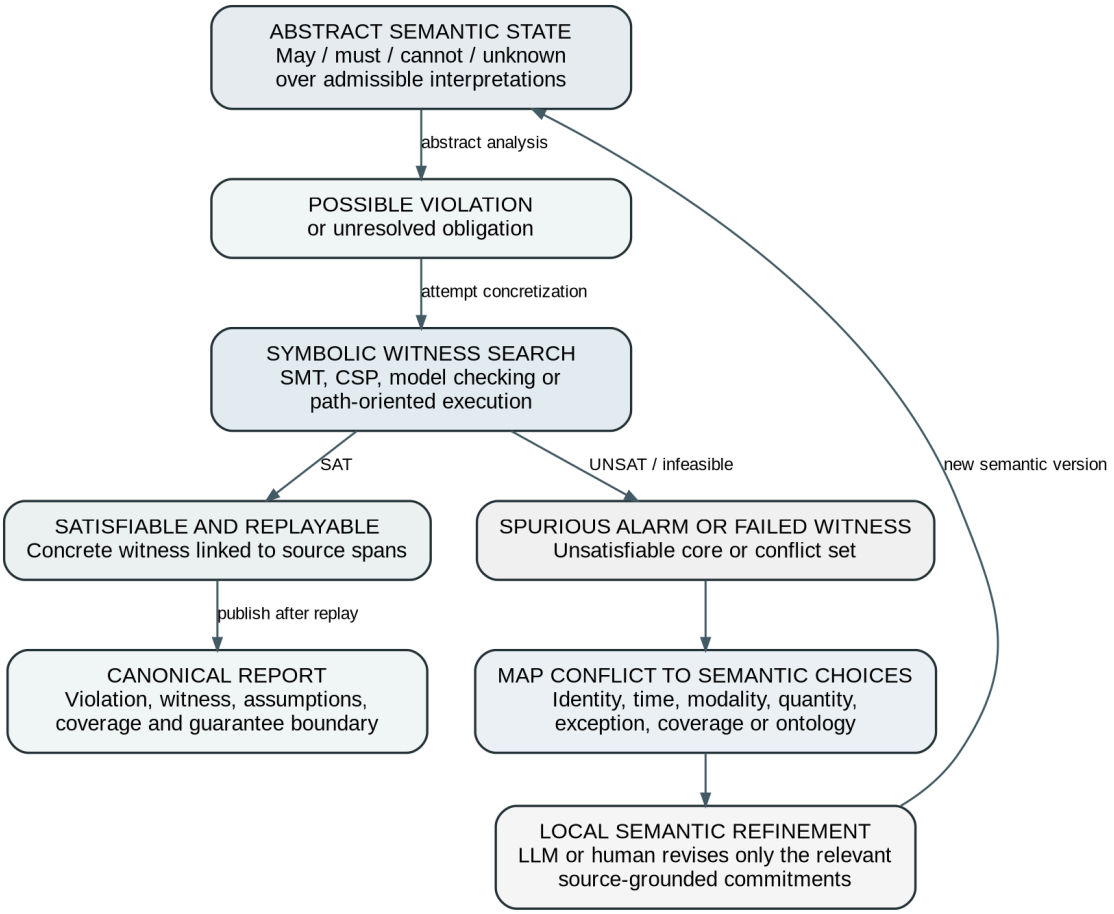


Figure 2. Counterexample-guided semantic refinement. Abstract analysis identifies a possible violation; symbolic witness search either confirms a replayable case or exposes the semantic distinctions that must be refined.

This leads to counterexample-guided semantic refinement. Traditional CEGAR begins with a coarse program abstraction. Model checking finds an abstract counterexample. A feasibility check determines whether the counterexample corresponds to a concrete execution. If not, predicates are added to refine the abstraction, and analysis repeats. In the language setting, the abstraction includes semantic choices. A spurious counterexample may arise because two mentions were merged, an exception was ignored, a temporal interval was widened, or a modal statement was treated as a fact.

A semantic refinement loop should preserve the symbolic explanation of failure. Suppose a compliance warning depends on the assumption that “the board” in two sections denotes the same body. Symbolic replay shows that the path becomes infeasible if the entities are distinct. The system maps the identity predicate to the relevant mentions and asks a focused question: are these references intended to denote the same board? An LLM can re-examine local context

and propose alternatives; a human can adjudicate if necessary. The rest of the document model remains unchanged. This is more reliable and efficient than asking a model to reread everything and “try again.”

Unsatisfiable cores provide a related mechanism. When a set of constraints is inconsistent, a solver can often return a subset responsible for the conflict. Mapping the core back to source spans identifies the minimum set of statements and interpretations that cannot coexist. In a policy review, this can reveal contradictory obligations. In semantic parsing, it can reveal that a quantifier, identity link, and temporal assumption jointly create impossibility. The user receives an explanation grounded in the formal conflict rather than a generic model critique.

Not every problem is best represented as path execution. Model checking analyzes transition systems against temporal properties. Linear temporal logic can express that something eventually happens, always holds, or occurs before another event. Computation-tree and probabilistic logics express branching or uncertain behavior. Natural-language-to-temporal-logic systems such as NL2TL and nl2spec show that LLMs can construct temporal specifications when the vocabulary and domain are controlled [Chen et al. 2023; Cosler et al. 2023]. The executor is then a model checker or planner, not an LLM.

Constraint solving covers a broad middle ground. SAT represents Boolean choices. SMT adds theories such as arithmetic, arrays, strings, and uninterpreted functions. CSP and CP-SAT handle finite domains, scheduling, and optimization. MaxSAT and soft constraints express preferences. Many natural-language tasks—scheduling, resource allocation, consistency checking, family relations, puzzles, and policy combinations—are better compiled into constraints than into procedural code. SatLM showed that declarative specifications plus a solver can outperform program-aided approaches on hard arithmetic and logical tasks [Ye et al. 2023]. The result supports a central architectural choice: the LLM should describe the problem, while the solver performs the search.

Formal planning provides action semantics and goal-directed search. Recent work positions LLMs as planning formalizers rather than planners. Hao and colleagues translate real-world planning constraints into formal verification tools, use unsatisfiable cores for repair, and report 93.9 percent success on TravelPlanner in their setting, compared with much lower direct-model baselines [Hao et al. 2025]. A 2025 survey concludes that the main opportunity is not replacing planners but using LLMs to construct and refine planning models [Tantakoun et al. 2025]. This is precisely the division advocated here.

Dataflow and Datalog engines provide fixpoint execution rather than path search. They are useful for monotone consequences and document-wide propagation. Automata and pattern engines handle ordered event sequences. Decision tables handle finite policies. The architecture should route each problem to the simplest formalism that expresses it. Symbolic execution should not become a universal label for every symbolic component.

LLMs can assist traditional symbolic execution in safer and riskier ways. A safer role is path prioritization. A model can identify code regions likely to contain vulnerabilities or paths likely to reach a target, while KLEE or another engine retains the final feasibility decision. Recent work such as KLEECopilot reports substantial coverage improvements from LLM-guided search, though the preprint status and model sensitivity require cautious interpretation [Chen et al. 2026]. The principle is sound: learned heuristics decide where to search, not whether a path is valid.

A riskier role is replacing the solver. AutoBug, presented by its authors as large-language-model-powered symbolic execution, decomposes code into path-based reasoning tasks and asks a model to reason directly about them without translating everything into SMT. The approach is lightweight and language agnostic, and reported experiments improve LLM-based program analysis [Li et al. 2025]. The authors characterize it as approximate. It can analyze constructs that are difficult for conventional solvers, but it does not provide classical soundness. For high-assurance use, such a model is better treated as one engine in a portfolio whose results require tests, replay, or independent confirmation.

LLMs can also generate path-satisfying tests, summaries, or models for external APIs. Studies report useful coverage on programs that traditional tools struggle to handle, but performance declines with long paths, complex APIs, recursion, and infeasible-path classification. These methods are promising because software environments contain semantics that are expensive to axiomatize. Their result status should remain “candidate” until concrete execution confirms the path or a solver proves the condition.

Security is a major concern. Natural-language rules and source documents are untrusted inputs. A malicious document can attempt prompt injection, fabricate references, or exploit a code-generating translator. Formal artifacts should be data, not unrestricted executable code. When custom algorithms are required, they should be reviewed host extensions with locked digests, typed inputs and outputs, resource limits, and no ambient authority. Solver invocations need time and memory budgets. A witness should include the exact version of the theory, compiler, and engine that produced it.

The combined analysis pattern is now clear. A coarse semantic and dataflow analysis identifies possible issues across a long document. Specialized formal engines test only the relevant slices. Symbolic execution constructs witnesses for path-dependent claims. Model checking handles temporal behaviors. Constraint solvers handle combinatorial choices. Failed witnesses and solver cores drive local semantic refinement. Every accepted conclusion is returned with a status and provenance that reflects the weakest link in the chain.

This architecture does not promise total verification of unrestricted language. It promises something more operational: every formal claim has an executable meaning; every symbol has a source or declared assumption; every counterexample can be replayed; every absence claim is tied to coverage; and every remaining ambiguity is represented rather than hidden. Abstract interpretation provides scale and conservative aggregation. Symbolic execution provides precision and witnesses. Their combination is one of the most plausible foundations for executable natural language.

4. What the LLM Era Has Actually Achieved

4.1 Autoformalization, Program-Aided Reasoning, Logic, Temporal Specifications, and Planning

The arrival of large language models changed the practical boundary of semantic compilation. Earlier systems required task-specific grammars, supervised logical forms, or carefully engineered lexicons. LLMs can propose formal artifacts from open-ended descriptions with little or no task-specific training. This substantially expands coverage, but it also creates a temptation to infer more from benchmark gains than the mechanisms justify. The relevant question is not simply whether a method improves answer accuracy. It is which errors have been transferred to a symbolic component, which errors remain in formalization, and what independent evidence exists that the formal artifact represents the source.

Program-Aided Language Models, or PAL, provide a clean starting point. The LLM reads a problem, generates a Python program, and delegates computation to the interpreter [Gao et al. 2022]. On GSM8K, the reported system improved substantially over chain-of-thought prompting. The mechanism is compelling because language models are often better at decomposing a problem than at carrying out long arithmetic exactly. Once the decomposition is correct, Python supplies deterministic operations and state.

PAL's limitation is equally instructive. The interpreter certifies only that it executed the generated program. It does not certify that the program faithfully models the natural-language question. A mistaken variable, omitted condition, or wrong formula can run without error. The system's success therefore depends on a distribution where programs are short, examples strongly constrain structure, and execution errors reveal many mistakes. For document reasoning, policy analysis, and scientific interpretation, the mapping is less constrained and the interpreter supplies weaker feedback.

SatLM replaces procedural code with declarative constraints [Ye et al. 2023]. The model states variables and relationships, and a satisfiability engine searches for a solution. This can be easier than generating an algorithm because the model need not decide the order of operations. Declarative representations also permit multiple models, infeasibility proofs, and constraint-level debugging. The reported gains on hard reasoning tasks support the idea that solver choice should follow problem structure. A schedule is naturally a constraint problem; a dynamic process may be a state machine; an arithmetic word problem may be either a program or constraints.

Logic-LM formalizes natural-language logic problems and invokes symbolic solvers [Pan et al. 2023]. It also uses solver errors to prompt self-refinement. Across several datasets, the authors report large average improvements over standard and chain-of-thought prompting. The architecture demonstrates two useful principles. First, the symbolic engine removes deductive inconsistency once a valid formalization exists. Second, formal error messages provide a better repair signal than a generic request to reconsider.

However, syntax repair is not semantic repair. A formula can be well formed, satisfiable, and wrong. Logic-LM's pipeline can correct malformed predicates or solver failures, but it may not detect that "some" became "all" or that an exception was omitted. Subsequent work such as Logic-LM++ adds multi-step refinement, and LTRAG retrieves thought-guided formalization

examples, reporting improvements over Logic-LM and LINC on FOLIO and AR-LSAT [Kirtania et al. 2024; Hu et al. 2025]. These gains suggest that formalization itself benefits from examples and structured critique. They do not eliminate the need to evaluate semantic fidelity directly.

LINC makes the modularity explicit: an LLM serves as a semantic parser, first-order logic represents the problem, and a theorem prover performs deduction [Olausson et al. 2023]. On ProofWriter, the reported combination produced large gains over chain-of-thought systems, including models substantially larger than the parser. Error analysis showed complementary failure modes: the symbolic system avoids many reasoning errors, while formalization failures remain. This is exactly the evidence expected from a federated architecture. The system is strongest when the natural-language task already corresponds to a compact first-order theory.

FOLIO provides a useful benchmark because it contains human-authored natural-language reasoning problems paired with verified first-order logic [Han et al. 2024]. It evaluates both reasoning and NL-to-FOL translation. The dataset remains challenging even for strong models, which undercuts the idea that first-order formalization is a solved preprocessing step. FOLIO also illustrates the importance of human-authored complexity. Synthetic rule datasets can make the language unusually regular and allow models to exploit templates. Human examples introduce lexical variation, world knowledge, and more realistic scope choices.

FoVer takes another approach by verifying natural-language reasoning through first-order logic [Pei et al. 2025]. Instead of treating fluent reasoning text as self-authenticating, it constructs formal obligations and checks them. This is a promising pattern for proof-constrained generation: an LLM may propose an argument, but a verifier accepts only inferential steps that correspond to valid formal consequences. The remaining issue is alignment between each sentence and the formal predicates used to check it.

Autoformalization research increasingly treats translation as a distinct discipline rather than a prompt trick. Surveys published in 2025 catalogue translation into theorem provers, logical languages, and formal specifications, emphasizing the need for feedback loops, retrieval, intermediate representations, and benchmarks that separate syntax from semantics [Weng et al. 2025]. Direct NL-to-FOL work reports meaningful progress but also systematic errors in implicit knowledge, quantifier scope, and ontology design [Yang et al. 2024; Lalwani et al. 2025]. Adding implicit background knowledge can improve task performance while increasing the risk that the formal system proves a premise not licensed by the source.

The treatment of background knowledge is a decisive architectural choice. In a puzzle benchmark, common-sense premises may be implicitly expected. In legal or scientific verification, introducing unstated premises can invalidate the result. A rigorous system should distinguish source premises, authority-pack premises, domain axioms, and model-suggested assumptions. The solver can reason over all of them, but the final explanation must show which conclusions depend on which authority. If an assumption is essential and unverified, the result should be conditional.

Temporal-logic translation brings different challenges. NL2TL uses lifted representations to learn temporal structure across domains and reports strong adaptation with relatively little target-domain data [Chen et al. 2023]. The lifting strategy hides concrete atomic propositions while learning reusable logical patterns. This is a valuable design insight: separate the structure of “always before,” “eventually after,” or “until” from the domain-specific grounding of events.

nl2spec emphasizes interaction [Cosler et al. 2023]. The system maps parts of the natural-language requirement to parts of the generated temporal formula, allowing users to repair local mistranslations. This interface acknowledges that ambiguity cannot always be solved automatically. It also suggests how LLM-based semantic refinement should work: the user should see the linguistic decision, not merely a string of temporal operators.

Grounding remains the fragile point. A temporal formula can have the correct shape while its atomic propositions refer to the wrong events or objects. Work on grounded logical equivalence reports that explicit grounding mechanisms improve correctness, reinforcing the distinction between formula syntax and semantic attachment. For an executable-language runtime, temporal atoms should be references to typed observations with spans and identities, not free-form labels invented by the translator.

Automated planning provides a particularly mature symbolic backend. Planning languages define types, predicates, actions, preconditions, effects, initial states, and goals. Planners then search for a valid action sequence. LLMs are poor long-horizon planners when they must maintain state and constraints internally, but they can be effective at formalizing a problem for a planner. A 2025 survey identifies model construction and refinement as the central role for LLMs in this area [Tantakoun et al. 2025].

Hao and colleagues demonstrate a rigorous version of the pattern on real-world planning [Hao et al. 2025]. The system formalizes constraints, invokes verification tools, and uses unsatisfiable cores to repair problems. The authors report 93.9 percent success on TravelPlanner in their experimental setting, far above direct LLM baselines they compare against. The result is important because it shows that formal tools can transform planning reliability when the problem is representable. It does not mean the LLM has become a sound planner. It means the planner and verifier carry the long-horizon state while the LLM constructs and repairs the formal instance.

Planning also exposes hidden constraints. Natural-language requests include commonsense expectations, preferences, and feasibility conditions that may not be stated as formal requirements. Systems such as LRPlan attempt collaboration between language and reasoning models for explicit and implicit constraints, while benchmarks such as PlanningArena show that even strong models struggle with tool selection, context memory, and logical consistency [Karthikeyan et al. 2025; Zheng et al. 2025]. A high-assurance system must either formalize the implicit constraints from a trusted source or label them as assumptions. “Commonsense” is not a sufficient provenance category.

RuleArena and related benchmarks test LLMs against realistic rule systems and find that tool augmentation helps but does not make formalization trivial. SemEval 2026 tasks on content and formal reasoning further show that models are influenced by semantic content even when logical form should determine the answer. Modular neuro-symbolic entries use FOL and Z3 to reduce content bias, but formal translation and multilingual noise remain difficult [Grimaldi 2026; Kartáč et al. 2026; Wang et al. 2026]. These results reinforce the need to disentangle linguistic interpretation from formal inference.

One of the most consequential observations across this literature is that successful systems use target-specific intermediate languages. PAL uses Python. SatLM uses declarative constraints. Logic-LM and LINC use logic. NL2TL uses temporal logic. Planning systems use PDDL-like models. SymGPT uses a grammar tailored to ERC rules. The field has not discovered a universal formal representation for natural language. It has discovered that LLMs

can route language into multiple formal ecosystems when examples, schemas, and feedback are available.

This supports a portfolio architecture rather than a universal “language bytecode.” A semantic pivot can preserve evidence, entities, events, scope, and modality, while a formalism selector chooses the execution target. The selector itself can be assisted by an LLM, but its decision should be validated against capability contracts. A formalism is compatible only if it can express the required distinctions and the necessary grounded observations exist. A planner should not be selected for a purely taxonomic query; first-order logic should not silently approximate uncertain temporal dynamics; a simple relational engine should not claim to decide defeasible legal exceptions.

The empirical evidence therefore supports a narrower but substantial claim. LLMs can dramatically improve the construction of executable formal objects. External symbolic systems can improve correctness, consistency, and interpretability. Solver feedback can guide repair. The strongest results appear in bounded domains with explicit languages, clear schemas, and evaluable answers. The unresolved challenge is semantic compilation: ensuring that the formal object preserves the source’s relevant meaning and that every premise is grounded and authorized.

4.2 LLM-Assisted Program Analysis, Symbolic Exploration, Abstract Transformers, and Certified NLP

The interaction between LLMs and program analysis runs in both directions. LLMs can help formal tools handle specifications, code patterns, and environments that are difficult to model. Formal tools can check code generated by LLMs, verify neural networks, and constrain learned components. This literature is especially valuable for executable natural language because it tests the same architectural question under clearer semantics: where can a probabilistic model contribute without becoming the final oracle?

Traditional static analysis depends on hand-designed abstractions, transfer functions, summaries, and environment models. Symbolic execution depends on path search and solver encodings. These tools offer principled guarantees but struggle with dynamic language features, library code, complex string manipulation, external services, and scalability. LLMs can recognize code intent, propose invariants, summarize functions, identify likely bug locations, and generate tests. A 2025 survey of LLM-assisted program analysis categorizes work into static, dynamic, and hybrid methods and emphasizes both the opportunity and the risks of probabilistic outputs [Wang et al. 2025].

One architectural role is heuristic guidance. A symbolic executor can retain its semantics and use an LLM to rank paths, identify vulnerable regions, or select loop bounds. If the heuristic is wrong, coverage may suffer, but accepted paths remain solver checked. This is analogous to using a neural model for search ordering in theorem proving. The distinction between guidance and acceptance is critical. A learned score may determine where resources are spent; it should not determine whether a path condition is satisfiable.

KLEECopilot represents this direction. The 2026 preprint uses LLM guidance to direct KLEE toward likely vulnerabilities and introduces heuristics for loop escape, reporting substantial improvements in basic-block and line coverage and dozens of unique violations [Chen et al. 2026]. As a recent preprint, the results require independent replication, and sensitivity to model family matters. Nevertheless, the design pattern is plausible: LLMs contribute semantic prioritization that classical search lacks, while KLEE retains the execution semantics.

Another role is generation of path-satisfying tests. A model can read code and propose inputs expected to traverse a target path. Concrete execution then validates whether the path was reached. This approach can handle API conventions or data formats that are burdensome for SMT. Studies report useful success rates but also failures on long paths, recursion, infeasible branches, and complex libraries. The safe interpretation is that the LLM is a test generator. A concrete run, instrumentation trace, or solver remains the validator.

AutoBug takes a more radical approach by using LLMs to perform path-decomposed reasoning directly [Li et al. 2025]. Instead of translating all constraints into a solver theory, it presents code-level path conditions to the model. This can be language agnostic and may analyze semantics that are hard to encode. The authors report improvements on several program-analysis benchmarks, particularly for smaller models. They also characterize the analysis as approximate. The system does not offer classical soundness, and its conclusions can vary with the model.

AutoBug is valuable because it shows where symbolic methods fail operationally. Many real programs call opaque libraries, manipulate complex objects, or depend on informal contracts. A language model can infer likely behavior from names, comments, and learned code patterns.

But this flexibility is the same source of uncertainty. In an assurance-oriented portfolio, approximate LLM analysis should generate candidates, summaries, or prioritized checks. Results should be confirmed by tests, contracts, runtime monitoring, or a formal engine whenever a safety claim is made.

LLMs can propose invariants for abstract interpretation. A candidate invariant can accelerate proof if a checker verifies that it holds initially, is preserved by transitions, and implies the property. This is a powerful division of labor: invariant discovery is creative and combinatorial, while invariant checking is mechanical. Similar patterns appear in theorem proving and program synthesis. The language model searches; the verifier accepts.

SAIL extends this principle to the synthesis of abstract transformers [Gu et al. 2026]. Constructing a transformer that is both sound and precise for a nonlinear operator is difficult. SAIL asks LLMs to generate candidates but imposes hard syntactic and semantic constraints and uses a mathematically grounded cost for unsoundness. The reported system synthesizes globally sound transformers across several domains, matches manual designs, and finds precise transformers not previously in the literature. The significance goes beyond neural-network verification. It is evidence that LLMs can automate formal-method construction when every accepted artifact is subject to independent proof obligations.

The contrast with asking an LLM to “act as an abstract interpreter” is sharp. Mitchell and colleagues prompted models to reason through abstract interpretation on SV-COMP programs [Mitchell et al. 2025]. Models could follow compositional or fixpoint styles on some cases but made logical errors and hallucinated on complex structures. The result is consistent with a general pattern: models can imitate formal processes and generate useful candidates, but reliability declines when the process requires persistent exact state. A real abstract interpreter should execute the lattice operations; the model can suggest domains, predicates, or explanations.

Formal verification of neural NLP models is another major branch. Certified robustness work asks whether a model’s prediction remains unchanged over all inputs in a defined perturbation set. Jia and colleagues used interval bound propagation for word substitutions and reported 75 percent adversarial accuracy on IMDB and SNLI under the constructed substitution families [Jia et al. 2019]. SAFER used randomized smoothing and could apply to black-box models including BERT [Ye et al. 2020]. Robustness-aware embeddings tightened certification bounds, improving certified robust accuracy on IMDB from 76.73 to 84.78 percent in the reported setting [Wang et al. 2023].

Transformer verification is difficult because self-attention contains products, softmax operations, and correlations across positions. Specialized domains improve bounds. Parameterized Abstract Interpretation for Transformer Verification introduced parameterized affine domains for inner products and reported tighter bounds and verification of instances that prior methods could not prove [Huang et al. 2026]. This line of work demonstrates that formal analysis can scale to nontrivial NLP architectures when the property and input set are precisely defined.

The scope of the guarantee must not be blurred. A certified classifier can be robust to every substitution drawn from a specified set and still be wrong. A substitution list may include contextually inappropriate “synonyms” or omit meaningful paraphrases. The certification says that the model is invariant over the formal perturbation family. It does not establish semantic

equivalence in human language. This distinction should be explicit in any executable-language system that uses certified extraction models.

A useful integration would attach a robustness certificate to a semantic observation. Suppose an event extractor identifies that a sentence states an obligation. A certified model might show that this classification is stable under a controlled family of lexical substitutions. The observation is still model proposed, but it has an additional stability property. Downstream symbolic analysis can record this evidence without upgrading the observation to fact. Such layered status is more informative than one confidence score.

Formal methods also verify code generated by LLMs. Test generation, static analysis, type checking, and proof assistants can detect many errors. Program synthesis systems increasingly use compiler and test feedback for iterative repair. The same pattern should govern generated semantic circuits and formalization code. A coding agent may author a rule implementation, but publication should require compilation, mutation tests, natural-language benchmark cases, replay, and independent verifiers. An LLM should not be allowed to create hidden executable behavior inside a declarative theory artifact.

The literature on protocol modelling from natural language offers a sobering result. Work presented at ICSE 2025 investigated generating Tamarin or ProVerif models from protocol documentation. Naive LLM use was ineffective; the successful pipeline required semantic decomposition, limited human disambiguation, and sound intermediate transformations. It generated correct moderate-scale models for 10 of 18 real protocols [Mao et al. 2025]. The result is meaningful because protocols contain roles, messages, cryptographic assumptions, state, and temporal order. It also shows that even strong structured documentation does not compile automatically into formal security models.

The failure cases are instructive. Documentation omits assumptions that experts infer. The same term can denote a message type, a state, or an instance. Security properties may be stated informally or distributed across sections. A formal protocol model must make adversary capabilities explicit. These are direct analogues of document reasoning. The solution was not a larger prompt but an architecture that separates semantic parsing, disambiguation, and verified transformation.

External reviews of the broader field converge on a similar conclusion. The 2026 neuro-symbolic NLP review favors stronger investigation of federative architectures but cautions that current comparisons are not controlled and that no foundational-scale federative system has been established [Chatzikyriakidis and Lappin 2026]. The planning-formalizer survey treats LLMs as constructors and refiners of formal models, not replacements for planners [Tantakoun et al. 2025]. The program-analysis survey emphasizes hybrid systems and the need to manage hallucination and reproducibility [Wang et al. 2025]. These are independent lines of evidence for the same division of labor.

The field has therefore achieved more than a superficial “LLM plus solver” pattern. It has demonstrated solver-guided repair, proof-checked formalizations, formal planning, path guidance, invariant proposal, transformer synthesis, and certified robustness. It has also exposed a stable boundary. When the LLM controls final acceptance, guarantees become approximate. When the LLM proposes artifacts that a separate formal system checks, strong guarantees can be retained for the checked property. The unanswered question is how to make the natural-language-to-artifact boundary comparably disciplined.

The architecture proposed later in this book applies program-analysis lessons directly. Candidate semantic transformations are treated like synthesized code. They have typed contracts, source maps, resource bounds, and tests. Abstract domains are registered rather than improvised during a run. Refinement predicates are suggested by LLMs but validated through concrete distinctions. Solver portfolios expose proof objects or witnesses. The language renderer consumes canonical results, not raw traces. In this design, the LLM's flexibility is an asset precisely because it is surrounded by symbolic acceptance gates.

5. Why Executable-Language Systems Fail

5.1 Grounding, Identity, Spurious Formalizations, Ontologies, and World Knowledge

The dominant error in executable-language systems is not an incorrect deduction from accepted premises. It is the admission of the wrong premises, objects, or relations. Grounding is the process by which symbols acquire operational reference: a predicate is connected to a phrase, an entity to one or more mentions, an event to evidence, a time expression to an interval, a database field to a concept, or an action to an environment capability. A formal program without reliable grounding is exact in the least useful sense. It executes consistently while describing the wrong world.

Grounding begins with source location. Every semantic observation should be linked to exact source evidence or explicitly marked as an external assumption. A quotation must be revalidated against the immutable source revision. A document summary cannot substitute for the source because it may omit qualifications. Retrieval results cannot support a claim about absence unless the retrieval process is complete for the relevant domain. This source discipline prevents a common failure in LLM systems: a model produces plausible supporting text that does not occur in the document, and downstream structured output gives the invention an appearance of authority.

Lexical grounding connects language to ontology. The phrase “approval” may denote an action, a state, a signed artifact, a permission, or an institutional decision. A domain schema must choose and document the interpretation needed by a rule. LLMs are useful because they can infer context and propose mappings, but a mapping should be tested against examples and alternatives. A string match between a phrase and a type name is not semantic compatibility. The nllAgent documentation explicitly warns against mapping two concepts merely because their labels match; it requires an adapter or semantic probe when meanings differ.

Identity grounding is more difficult and more consequential. Document analysis often needs to know whether two mentions denote the same person, organization, object, dataset, requirement, or event. Equal names do not imply identity, and different names do not imply difference. Titles change, aliases occur, pronouns are ambiguous, organizations have subdivisions, and fictional or quoted mentions may not belong to the source world. A safe identity layer separates mentions from entity hypotheses and identity candidates. It records the producer, evidence, scope, world, and alternatives for each link.

Premature identity merging creates cascading errors. In a narrative, two characters with the same surname may be merged, producing impossible continuity violations. In a regulatory document, “the Authority” may refer to different bodies in different jurisdictions. In software documentation, “client” may denote a user, an application component, or an HTTP endpoint. Once merged, later rules treat all properties as belonging to one object. The error becomes difficult to diagnose because the solver sees a consistent identifier. A conservative system should allow unresolved identity and return unknown when a rule requires a link that has not been established.

Temporal grounding has similar layers. Dates can be explicit or relative. “Within thirty days” requires an anchor event and a calendar policy. “Before release” may refer to a planned or

actual release. “After approval” may mean immediately after or at any later time. Jurisdictions define business days differently. A temporal solver can manage intervals and ordering exactly once these choices are made, but it cannot infer the intended anchor from an arbitrary label. Temporal observations should therefore include the textual expression, normalized value or constraint, anchor, calendar, timezone, confidence, and alternatives.

Normative grounding is frequently mishandled. “Must,” “shall,” “should,” “may,” “can,” and “is expected to” have domain-specific force. Legal drafting conventions differ from ordinary language. An exception may defeat a rule, narrow its scope, or create a separate permission. Conflicting authorities may have priorities. A simple Boolean representation of compliance loses these distinctions. Deontic logic, rule priorities, decision tables, or defeasible systems can model them, but the source-to-rule mapping remains a linguistic and institutional judgment.

Spurious formalizations arise when the wrong program produces the expected result. Weakly supervised semantic parsing made this problem explicit, but it is pervasive in LLM evaluation. A generated formula may solve a benchmark because examples have regular templates. A text-to-SQL query may return the correct current rows despite a wrong join. A planning model may find a plan because an omitted constraint is irrelevant in the tested instance. A rule circuit may pass ten examples while encoding the wrong concept. Final-answer accuracy alone cannot distinguish these cases.

The appropriate defence is semantic mutation. The evaluator changes one aspect of the source or environment while preserving others, then checks whether the formalization changes in the expected way. If “all” becomes “some,” the quantifier should change. If an exception is removed, the previously exempt path should become prohibited. If two entities are renamed to the same string, identity should not merge automatically. If the database instance changes, an accidentally correct SQL query should fail. Mutation testing evaluates the meaning of the program rather than only its behavior on one fixture.

Counterexample generation can compare competing formalizations. Given two logical or executable programs, a solver can search for a model in which their outputs differ. The resulting scenario is translated into natural language and presented to a reviewer. This turns an opaque formula comparison into a concrete question: “Under interpretation A, a supervisor who has not completed training may approve; under interpretation B, they may not. Which is intended?” The method is especially useful when several LLMs produce plausible but non-equivalent forms.

Ontology design creates another failure mode. A universal schema is attractive for reuse, but it can hide domain disagreements and force concepts into inappropriate categories. An extremely narrow schema, by contrast, produces many incompatible local languages. A layered design is preferable. The upper layer defines source, evidence, identity hypotheses, worlds, time, modality, status, coverage, and gaps. Domain packages define substantive types and relations. Mapping packages declare correspondences, losses, and assumptions between domains.

Ontology compatibility should be operational, not rhetorical. A circuit declares the types and guarantees it consumes. A producer declares which types it can materialize, under which statuses and coverage modes. Publication checks whether a compatible producer exists. Runtime checks whether the concrete document yielded the required observations. If the circuit needs human-confirmed identities but only model-proposed identity candidates exist, it cannot run at the claimed guarantee. If a temporal rule needs a jurisdictional calendar that is absent, the result is an operational gap, not a guessed date.

World knowledge is particularly hazardous. LLMs contain broad knowledge but no stable provenance, effective date, or jurisdiction. Treating model memory as a formal fact base creates invisible dependencies and stale conclusions. A symbolic system should load explicit knowledge packs with source identifiers, digests, effective intervals, jurisdictions, conflict policies, and guarantee ceilings. The pack may be curated, retrieved, or model-assisted, but its accepted claims are external artifacts. A legal pack can expire. A scientific pack can be superseded. A fictional-world pack can deliberately contradict current reality.

Conflicts between sources should not be erased through majority voting unless the task defines such a policy. The system may need paraconsistent reasoning, source priority, temporal versioning, or separate worlds. A claim can be supported by one source and contradicted by another. The canonical result should preserve both and state which policy, if any, resolves them. This is essential for scientific surveys and regulatory analysis, where disagreement is information rather than noise.

Background assumptions must be visible. Logic benchmarks often assume that names denote distinct individuals or that categories are disjoint. Real documents rarely state all such axioms. An LLM may insert them because they make the puzzle solvable. In a research or compliance system, those assumptions should appear in a separate authority layer. The final answer can then state that a conclusion follows only if the assumption holds. This prevents implicit world knowledge from masquerading as document evidence.

Coverage is the grounding of absence. To conclude that a document does not contain a required approval, the system must know what was searched, which sections and channels were included, how the concept was recognized, and whether the search method was complete for the claim. A lexical scan may be complete for an exact phrase but not for paraphrases. Retrieval may be high recall but not closed-world. A human-curated registry may be complete for a release. Coverage should be a first-class object consumed by negative rules.

The nlIAgent architecture contains an important version of this principle. LongTextJS records coverage and gaps alongside observations. Circuit compatibility checks whether the actual program satisfies required status and coverage. Missing knowledge is not reported as compliance. This is stronger than the common pattern of asking an LLM to return `true`, `false`, or `not found`, because `not found` is interpreted relative to a documented search domain [AssistOS-AI 2026].

The remaining difficulty is how to validate model-produced observations. Schema validation is necessary but weak. Evidence anchoring prevents invented quotations but not misinterpretation. Calibration scores help rank candidates but do not establish meaning. The strongest practical approach combines independent signals: controlled-language parsers for tractable fragments; model ensembles or diverse prompts; ontology constraints; cross-sentence consistency; semantic mutations; negative examples; human review for high-impact distinctions; and downstream counterexamples that expose consequences.

Grounding should also be query driven. A system need not resolve every entity or event in a 300-page document. It should derive semantic demand from the rule being executed, retrieve the relevant scopes, and materialize only the necessary types. However, query-first processing must not confuse relevance with completeness. If a rule asks whether any exception exists anywhere in the authority text, retrieval must provide a coverage argument or the answer remains unknown. Demand-driven execution reduces work; it does not justify premature closure.

The fundamental conclusion is severe but constructive. There is no purely symbolic remedy for a symbol attached to the wrong referent. Grounding is not a preliminary NLP detail; it is part of the trusted computation. The system must expose the chain from source to symbol, preserve alternatives, track coverage, and refuse incompatible execution. Once that chain is disciplined, mature formal tools can add substantial reliability. Without it, the tools merely certify the consequences of a plausible fiction.

5.2 Pragmatics, Nonmonotonicity, Scale, Security, and Evaluation Failure

Even a well-grounded representation can fail because natural language does more than state monotone facts. Speakers imply, presuppose, revise, quote, joke, negotiate, and rely on institutional context. Documents contain exceptions, priorities, definitions that change later, and statements whose force depends on genre. Executable natural language must decide which of these phenomena it intends to support and must return unknown outside that scope.

Pragmatics is the first boundary. A sentence can communicate more than its literal truth conditions. “The report is almost complete” may imply that it is not complete. “Some tests passed” can pragmatically suggest that not all did, although classical logic does not entail that. Irony reverses apparent sentiment. Indirect requests use statements as actions. A formalizer that treats pragmatic inferences as facts can overreach; one that ignores them may miss the author’s practical intent. The appropriate treatment depends on the application. A compliance engine should usually privilege explicit authority and label pragmatic interpretation as proposed. A conversational agent may need richer pragmatic models but should not represent them as hard evidence.

Presupposition creates a related challenge. “The committee stopped reviewing the proposal” presupposes that it had been reviewing it. Definite descriptions often presuppose existence or uniqueness. Documents can exploit such constructions without explicitly asserting the background. A formal system must decide whether presuppositions enter as assumptions, defeasible inferences, or requirements for clarification. Automatically asserting them can be unsafe; ignoring them can make discourse incoherent. The status system should preserve their origin.

Natural-language rules are often nonmonotonic. New information can retract a conclusion. “Employees may access the system” may be defeated by “unless their certification has expired.” Default logic, answer-set programming, defeasible logics, and prioritized rules provide formal mechanisms for exceptions. Classical first-order logic can encode exceptions explicitly, but large rule sets become difficult to manage and inconsistent information can trivialize inference if ordinary explosion applies.

A practical runtime should support multiple rule semantics. Monotone Datalog is appropriate when facts only accumulate. Defeasible rules are appropriate for defaults and exceptions. Paraconsistent reasoning is appropriate when sources conflict and the system must avoid deriving everything. Decision tables are appropriate when policy combinations are finite and explicit. The chosen semantics must be versioned with the rule package; it cannot be inferred ad hoc by the LLM during execution.

Normative systems add authority and priority. A later rule may override an earlier one. A higher-level regulation may pre-empt a local procedure. Jurisdiction and effective date matter. Permissions and obligations can conflict. Deontic paradoxes show that naive logical encodings can produce counterintuitive results. Many practical cases are better handled by explicit rule priorities, exceptions, and violation conditions than by a universal deontic calculus. Executable natural language should favour operational clarity over theoretical grandeur.

Scale creates several interacting explosions. Long documents contain many spans and candidate observations. Ambiguous mentions create combinatorial identity assignments. Temporal relations create partial-order alternatives. Rules create derived consequences.

Symbolic execution creates paths. Solvers face large constraint systems. LLM context windows and costs impose separate limits. A naive architecture that formalizes every sentence into every possible logic is infeasible.

Demand-driven materialization is the first response. Circuits declare what they consume. The runtime indexes structure and stable lexical information, then produces richer semantics only for relevant scopes. Incremental computation reuses unchanged artifacts. Abstract interpretation summarizes alternatives. State merging and bounded symbolic exploration control paths. Solver portfolios route simple cases to cheap engines. Hard budgets produce explicit incomplete results rather than hidden truncation.

The second response is semantic modularity. A document can be analysed through views: sections, time windows, jurisdictions, narrative scenes, experimental stages, or authority layers. Identity and rule consequences may be local to a view. Cross-view relations are introduced only when required. This reduces accidental global merging and supports parallelism. The system should still record when a conclusion depends on the assumption that a view is complete.

The third response is staged precision. An inexpensive structural or lexical pass eliminates many cases. A model-assisted extraction pass produces candidates for the remaining ones. Abstract analysis identifies potential violations. Symbolic refinement handles only result-relevant ambiguities. Human review is reserved for high-impact unresolved choices. This resembles static-analysis pipelines that use cheap analyses before expensive theorem proving.

Security threats arise at every stage. Prompt injection in source documents can instruct the LLM to ignore schemas or fabricate results. The translator should receive source content as data under a role-specific prompt, and its output should be validated independently. The source must never gain authority to modify the runtime, rule package, or publication policy. Any text claiming to be a system instruction is still document content.

Generated code is an even greater risk. A DSL that allows arbitrary JavaScript, Python, imports, or network access turns semantic learning into code execution. Declarative circuits should normalize to plain data and call only registered operators. Custom algorithms should be installed as reviewed extensions with typed contracts, effects, resource limits, and locked digests. Sandboxing remains necessary, but capability restriction is stronger than relying only on a process boundary.

Solver attacks include denial of service through pathological constraints, malformed proof objects, and exploitation of native libraries. Budgets, theory restrictions, proof checking, and version pinning are required. A model can deliberately or accidentally construct a formula that is technically valid but computationally explosive. Formalism selection should estimate cost and reject tasks outside policy.

Data poisoning affects learned semantic producers and benchmark suites. If a coding agent generates both a circuit and the examples that validate it, the artifacts may share the same misunderstanding. Independent test generation, semantic mutations, manually curated anchor cases, and separate review roles reduce this risk. The nllAgent learning architecture correctly prevents a coding agent from publishing its own candidate. Manual publication is a governance control, although its effectiveness depends on the quality of benchmarks and independent checks.

Natural-language realization can reintroduce unsupported claims. A model may convert “possible under one interpretation” into “the document violates the rule.” It may omit

assumptions or turn an incomplete search into absence. Realization should therefore be constrained by a canonical result schema and checked at claim level. High-impact outputs may use controlled templates. Explanatory commentary should be clearly separated from verified findings.

Evaluation is perhaps the most systemic failure. Many benchmarks measure only final answer accuracy. This confounds extraction, formalization, reasoning, and rendering. A model may reach the correct answer for the wrong reason. A formal system may be penalized for returning unknown where the benchmark assumes an unstated convention. Synthetic datasets may reward template recognition. LLM-as-a-judge metrics can share biases with the systems being evaluated.

A rigorous benchmark should contain multiple layers. The source layer includes exact text and structure. The semantic layer includes one or more admissible interpretations with evidence spans. The formal layer includes target programs or extensional equivalence tests. The execution layer includes expected models, proofs, witnesses, or unsatisfiable cores. The result layer includes canonical statuses and coverage. The realization layer includes allowed claims and forbidden overstatements. Errors can then be attributed rather than collapsed into one score.

Minimal pairs are essential. Changing one word or relation should produce a predictable formal difference. Paraphrase families test invariance. Distractor sections test retrieval. Identity collisions test grounding. Missing evidence tests unknown behavior. Conflicting sources test paraconsistency or authority policy. Temporal boundary cases test calendars and intervals. Adversarial prompt text tests security isolation. Semantic mutations test whether a circuit actually implements its rule.

Evaluation should also measure abstention. A trustworthy system must know when it lacks an ontology, producer, coverage, or budget. Unknown is not a failure when the question is unsupported. Metrics should distinguish justified abstention from avoidable incompleteness. Coverage-quality curves can show how accuracy and false assurance change as the system admits more model-proposed semantics.

Independent implementation matters. A circuit compiler and its “shadow” evaluator should not share every critical function if differential testing is claimed. Mutation scores should come from executable mutations, not prose descriptions. Live model profiles should be evaluated separately from deterministic fixtures. Planning and realization need task-specific benchmarks rather than being inferred from audit performance. These are precisely several of the serious issues acknowledged in the nllAgent repository [AssistOS-AI 2026].

External reviews should be interpreted with similar care. The Frontiers 2026 survey is peer reviewed and useful, but its claim that federative systems perform better is explicitly qualified by task confounding. Planning and program-analysis surveys map trends but cannot establish that a proposed architecture works on unrestricted documents. No independent review of nllAgent was found. A responsible book should therefore use reviews to identify consensus and disagreement, then state its own judgments where evidence is absent.

The central evaluation principle is causal attribution. When a system improves, researchers should determine whether the gain came from better semantic parsing, solver correctness, retrieval, test-time sampling, or benchmark leakage. When it fails, they should identify the layer.

This is not merely scientific hygiene. Layered error data is what allows the architecture to improve without replacing the entire system or hiding regressions in aggregate scores.

Executable natural language will remain a bounded capability. Pragmatics, open-world knowledge, social context, and evolving institutions ensure that no finite symbolic kernel captures all meaning. The goal is not to remove this boundary but to make it explicit and useful. A system can offer strong guarantees for structural, relational, temporal, quantitative, and rule-governed fragments while returning conditional results elsewhere. The most dangerous system is not one that admits limits. It is one that converts unmodelled language into confident symbolic output and calls the result verified.

6. Architecture Families for Executable Natural Language

6.1 Competing Designs: Universal Representations, Formalism Portfolios, Controlled Compilation, and Query-First Systems

Executable-language architectures can be organized according to where they place semantic commitment. Some attempt to translate every input into one universal representation. Others retain several candidate forms and select a formalism per task. Some restrict the input language until compilation is deterministic. Others preserve open language but formalize only the observations demanded by a query. Each design solves a different problem, and a mature system will likely combine them rather than select one exclusively.

The universal intermediate representation is the most intuitive architecture. Text is compiled into a rich semantic graph or logical language, and every downstream operation queries or transforms that common object. The advantages are conceptual unity, reuse, and the possibility of global consistency checks. A single representation can support multiple applications and create a stable interface between NLP and reasoning.

Its weakness is semantic overreach. To serve every task, the representation must cover entities, events, discourse, time, modality, quantity, causality, norms, uncertainty, provenance, and domain-specific concepts. Either the language becomes extremely complex or it simplifies distinctions that some task needs. A universal representation also creates a single point of failure: a wrong global parse contaminates every analysis. The pressure to choose one identity, scope, or ontology early can hide ambiguity.

A semantic pivot is a more modest variant. It does not claim to be a complete logical form. It standardizes source spans, mentions, entity hypotheses, observations, worlds, time-bearing values, modality, provenance, alternatives, coverage, and gaps. Specialized compilers lower selected portions into formal targets. This preserves common infrastructure without pretending that one execution semantics fits all meaning. LongTextJS, as documented in nllAgent, is close to this idea: it is an addressable source world and observation program rather than a universal theorem-proving language [AssistOS-AI 2026].

The multiple-candidate architecture treats formalization as search. An LLM produces several logical forms, programs, or semantic graphs. Type checks and ontology constraints eliminate invalid candidates. Execution, examples, and counterexamples rank or distinguish the survivors. The system can return a must conclusion when all candidates agree and a may conclusion when only some do. This architecture is appropriate when ambiguity is genuine or when formalization reliability is limited.

Its cost is combinatorial. Candidate forms multiply across sentences and identity choices. A naive cross-product becomes infeasible. The solution is to represent shared structure and local alternatives compactly, use abstract domains to summarize them, and refine only result-relevant choices. Confidence can guide exploration, but low-probability candidates should not be removed from a sound analysis unless the policy explicitly accepts probabilistic incompleteness.

Controlled-language compilation takes the opposite approach. The input is rewritten into a restricted natural language whose grammar and semantics are deterministic or highly

constrained. Users can author directly in the controlled language, or an LLM can propose a controlled paraphrase for confirmation. Once accepted, compilation is reliable. This architecture is particularly strong for requirements, contracts, procedures, and rule authoring.

Its limitation is coverage and adoption. Ordinary documents contain constructions outside the controlled fragment. Rewriting can change meaning. Users may approve a paraphrase without noticing a subtle difference. The architecture works best when the controlled form is displayed alongside the source, differences are localized, and formal counterexamples make important choices concrete. Controlled language should be an explicit semantic checkpoint, not an invisible normalization.

Query-first architectures avoid total formalization. A circuit or question declares which observations and relations it needs. The runtime preserves the source structurally, retrieves relevant regions, and materializes only demanded semantic types. Relational queries, tables, and decision rules operate over the resulting finite world. This design scales better to long documents and aligns computation with purpose.

A closely related 2026 database-systems vision distinguishes interpreted semantic operators from compiled ones. In the interpreted design, an LLM is invoked inside the data-processing loop for each row, record, or candidate pair. In the compiled design, the LLM is invoked during a separate construction phase to translate the semantic operation into deterministic local code that can be executed repeatedly without a model call. Preliminary experiments reported substantial reductions in latency and model use while retaining much of the output quality [Dong and Wang 2026]. The proposal is important because it independently reaches the same architectural intuition developed in this book: the language model is most useful as a compiler or plan constructor, while the repeated runtime should be stable and inspectable. Its limitation is equally instructive. Generated code may approximate the model-defined operator efficiently without supplying a semantic proof that it captures the user’s intended concept. Compilation removes per-item stochasticity; it does not by itself solve grounding or specification fidelity.

The risk is false closure. Retrieval may miss relevant evidence, and demand analysis may omit a needed concept. Query-first systems must track coverage and producers. If a negative conclusion requires searching the whole document for any exception, a few retrieved passages are insufficient. The architecture should distinguish a bounded query result from a closed-world guarantee. nllAgent’s compatibility and coverage contracts are an important mechanism in this family.

A single-formalism architecture lowers all tasks into one logic, often first-order logic, Datalog, or SMT. It simplifies implementation and proof obligations. For a narrow domain, this can be the best choice. Logic-LM and LINC gain much of their value from a stable target language. A system for scheduling may reasonably use one constraint language. A protocol verifier may use one process calculus.

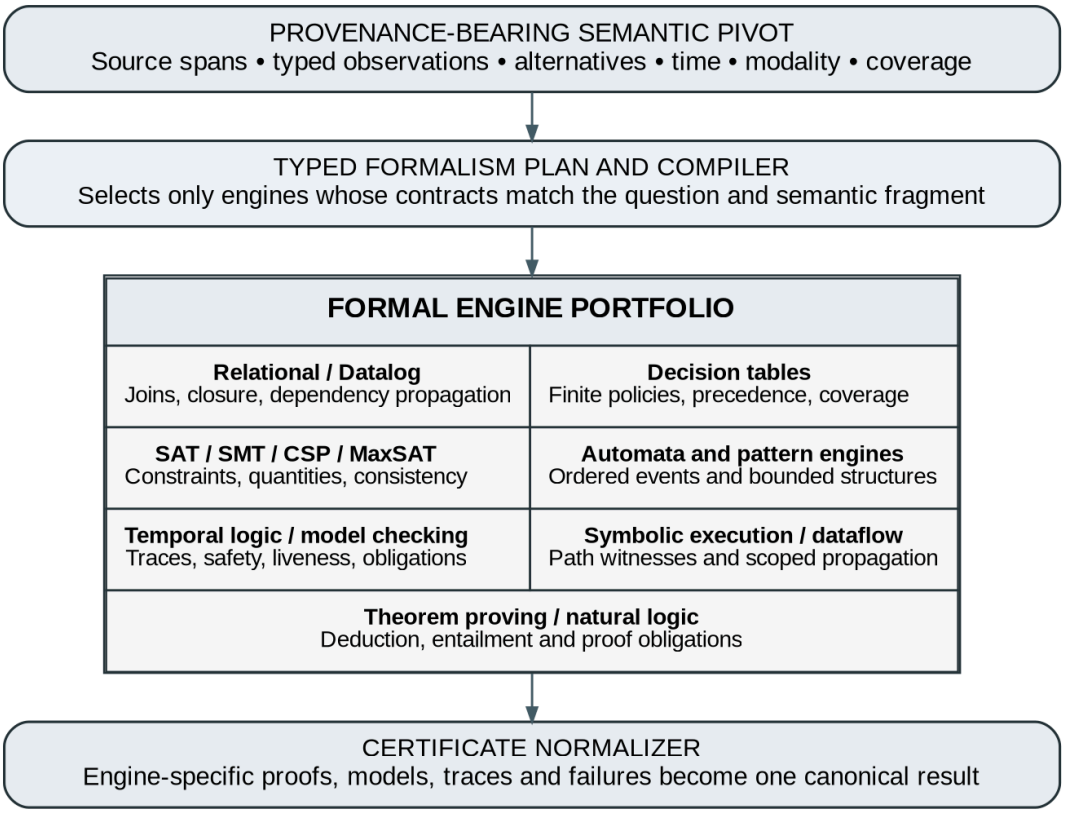


Figure 3. A provenance-bearing semantic pivot can be compiled into a portfolio of specialized formal engines and normalized into one canonical result.

For broad natural-language tasks, a solver portfolio is more plausible. Relational evaluation handles document queries. Datalog computes monotone closure. Decision tables express finite policies. SAT and SMT handle constraints and arithmetic. Model checking handles temporal behavior. Symbolic execution constructs path witnesses. Natural logic handles certain entailment relations near surface language. Automata recognize ordered patterns. The semantic pivot supplies grounded atoms, and a formalism plan selects one or more engines.

The portfolio creates new obligations. The router must know the expressive and operational capabilities of each engine. A compiler must preserve meaning across lowerings. Results from different engines need a common status and provenance model. If two engines reason over overlapping predicates, consistency must be checked or explicitly left open. Resource policies must prevent a model from selecting an unnecessarily expensive solver.

A useful formalism contract states the input types, supported operators, semantics, required coverage, guarantee class, expected certificate, cost model, and known limitations. A temporal-logic engine may require grounded events and a finite transition system. An SMT engine may support integer arithmetic but not unrestricted nonlinear real functions. A decision table may require finite enumerated conditions. A natural-logic engine may support monotonicity but not arbitrary cross-sentence coreference. The router can then reject a proposed plan whose requirements exceed the engine.

Proof-carrying generation is another architecture family. The primary artifact is not the natural-language answer but a canonical result plus a proof, witness, execution trace, or verifiable derivation. The language model generates an explanation from that artifact. Every

explanatory claim is linked to a result element. An independent checker ensures that the explanation does not exceed the formal content.

This design is valuable because generation and verification have different strengths. An LLM can write a clear explanation of a counterexample or unsatisfiable constraint. A symbolic result can ensure that the scenario is real under the accepted model. The explanation may still be misleading if it hides assumptions or uncertainty, so the canonical result must include those dimensions and the renderer contract must require them when material.

Abstract-semantic architectures place an abstract interpretation layer before specialized execution. Candidate observations populate may and must domains. Cheap propagation determines which properties are definitely satisfied, definitely impossible, or potentially violated. Only potential violations are lowered into expensive symbolic problems. This reduces solver load and provides principled unknown states. It also creates a natural incremental architecture for document edits.

Counterexample-guided architectures close the loop. A failed proof or spurious witness is not merely an error; it identifies which semantic distinction matters. The system maps solver facts to source spans, asks the LLM or user to refine the relevant interpretation, and replays the analysis. This architecture is especially promising because it transforms formal feedback into targeted language understanding rather than generic self-reflection.

Learning architectures govern how new rules, ontologies, and circuits enter production. One extreme allows an LLM agent to generate executable code during every run. This maximizes flexibility and minimizes assurance. The opposite extreme permits only hand-written, formally verified rules. A practical middle uses coding agents in staging workspaces, generates benchmarks and mutations, restricts artifacts to declarative forms, and requires explicit publication. The production runtime reads immutable releases. This is the governance model implemented in nlIAgent and should be generalized.

The following comparison summarizes the core trade-off without claiming that one design dominates every task.

Table 1. Main architecture families and their essential trade-offs.

Architecture	Essential strength and principal risk
Universal semantic representation	Reuse and global consistency; risks overcomplexity and premature semantic commitment.
Semantic pivot plus specialized lowerings	Preserves common evidence and supports multiple engines; requires careful compiler and cross-engine contracts.
Multiple candidate formalizations	Represents ambiguity and enables may/must conclusions; risks combinatorial growth.
Controlled-language compilation	Strong semantics and user-inspectable forms; sacrifices coverage and may introduce meaning changes during rewriting.
Query-first materialization	Scales to long documents and task-specific needs; can confuse retrieval with completeness unless coverage is explicit.
Single formal target	Simple trust base and evaluation in a bounded domain; poor fit for heterogeneous language phenomena.
Solver portfolio	Matches computational method to problem structure; adds routing, interoperability, and certificate-normalization obligations.

Architecture	Essential strength and principal risk
Proof-carrying generation	Constrains natural-language answers to verified content; requires robust claim-level realization checking.
Abstract analysis plus symbolic refinement	Balances scale and precision; soundness remains conditional on interpretation coverage.
Governed learned circuits	Permits evolution without run-time self-modification; depends on strong benchmarks, mutation testing, and publication review.

The most credible general architecture is a hybrid of the middle options. It uses a provenance-bearing semantic pivot, preserves alternatives where they matter, performs query-driven materialization, routes to a formalism portfolio, and accepts only proof- or witness-backed results. Controlled language can serve as an interactive checkpoint for high-impact formalizations. Abstract interpretation controls scale, while counterexample-guided refinement improves precision. Learning remains outside production.

This conclusion is not only theoretical. It is reflected in the empirical literature. PAL, SatLM, Logic-LM, LINC, temporal-logic translators, planning formalizers, SymGPT, and certified-verification systems all succeed by narrowing the formal target. No result demonstrates that one universal logic or one end-to-end model can replace the portfolio. The architecture should therefore embrace heterogeneity while minimizing the trusted interfaces between components.

6.2 A Reference Pipeline from Text to Symbols, Execution, and Constrained Natural-Language Answers

A reference executable-language pipeline should be understood as a compiler and verification toolchain rather than an agent prompt. Each stage materializes an immutable or content-addressed artifact. A run can be replayed without asking the model to reconstruct its previous reasoning. Errors are assigned to stages. Learning can improve a producer or theory without silently changing the meaning of old results.

The first artifact is the source package. It contains the canonical text or data, revision digest, language, structure, channels, and stable spans. Paragraphs, headings, tables, quotations, and code blocks retain order and identity. External records have similarly stable keys. The source package is lexical authority: it establishes what was present, not what it means.

The second artifact is the task contract. It states the question, rule release, source scope, desired guarantee, allowed assumptions, world and jurisdiction, budgets, and output policy. The task determines semantic demand. A request to find undefined acronyms requires different observations from a request to verify temporal compliance. The contract also defines whether absence can be concluded and which forms of uncertainty require human review.

The third stage constructs an evidence graph. Deterministic producers add structural observations. Controlled-language parsers add observations for the fragments they recognize. LLM-backed producers add schema-bound candidate events, relations, identities, modalities, or quantities. Every observation records anchors, producer, model or algorithm version, status, alternatives, and coverage. The model does not write directly into the canonical graph; a materializer rechecks spans, types, and referential integrity.

Identity and world management occur in the same layer but are not silently resolved. Mentions are local evidence. Entity hypotheses collect mentions under a world and scope. Identity candidates state same, different, or unresolved relations with support. Mutually exclusive interpretations belong to separate worlds or guarded alternatives. The graph can therefore represent disagreement without corrupting global consistency.

The fourth stage computes an abstract semantic state. Domain-specific abstractions summarize possible identities, temporal orders, normative statuses, quantities, and evidence levels. Must facts hold across all admitted worlds; may facts hold in at least one. Unknown records a gap rather than a probabilistic guess. The abstract state supports cheap rule propagation and identifies which questions require greater precision.

The fifth stage produces a formalism plan. The planner is not a free-form LLM decision. It compares the task with registered formalism contracts. A simple document query may remain relational. A policy may compile into decision tables and constraints. A procedure may compile into a transition system. A path-dependent violation may invoke symbolic execution. A deductive claim may invoke a theorem prover. The plan names required inputs, lowerings, engine versions, resource bounds, and expected certificates.

The LLM can propose a plan because it recognizes problem structure and domain language. A deterministic checker validates that the plan uses supported formalisms, that every required observation has a compatible producer, and that no semantic distinction essential to the task is lost. For example, a rule involving exceptions and priority cannot be compiled into a monotone

engine without an explicit encoding. A rule involving identity cannot proceed if the identity domain is unresolved and the verdict depends on it.

The sixth stage performs typed compilation. This compiler should be small, deterministic, and heavily tested. It converts semantic objects into solver variables, relations, transitions, or predicates. Source maps connect every formal element back to observations and spans. Compilers should generate proof obligations when a lowering is lossy or depends on assumptions. Different compilers can be compared through semantic mutations and distinguishing models.

The seventh stage is formal execution. Relational and Datalog engines compute closures. Decision tables identify matched and conflicting rows. SAT/SMT/CSP solvers find models or prove infeasibility under their supported theories. Model checkers evaluate temporal properties. Symbolic executors explore path conditions and return concrete witnesses. Proof assistants or theorem provers produce derivations. The engine output is not yet a user-facing verdict; it is a certificate-bearing execution record.

The eighth stage is replay verification. A separate verifier checks the witness or proof against the source-linked formal problem and the published theory. For a document finding, it confirms that the cited observations exist, spans match the source revision, statuses satisfy the rule, and the execution trace supports the claim. For a negative result, it confirms that coverage and completeness assumptions are sufficient. Replay is valuable even when the underlying solver is trusted because it validates the connection between candidate output and released policy.

If abstract analysis predicted a violation but symbolic execution cannot realize it, semantic refinement begins. The system extracts the unsatisfiable core, conflicting predicates, or proof failure. Source maps identify the relevant spans and semantic choices. The LLM receives a bounded task: reconsider this identity link, temporal scope, exception mapping, or predicate alignment. It proposes revised candidates; the materializer and compiler revalidate them; execution repeats. Human review is requested when the distinction is high impact or remains unresolved.

This loop should be monotone with respect to evidence, not merely confidence. A refinement can split an entity, narrow a time interval, add an overlooked exception, or mark a premise unsupported. It should not upgrade a proposed observation to fact because the new result is convenient. Every change is recorded as a new semantic revision. Previous paths remain reproducible.

The ninth artifact is a canonical result. It contains the rule or query identity, accepted status, assumptions, must and may conclusions, rejected candidates, unknowns, coverage, source evidence, proof or witness references, and execution bounds. It may also include a concise machine-checkable explanation graph. This artifact is authoritative over any prose rendering.

The tenth stage realizes natural language. The renderer receives only the canonical result and selected source excerpts. It does not rerun the original reasoning. Each generated sentence is associated with one or more result claims. A checker compares the sentence with the allowed claims, ensures that modality and uncertainty are preserved, and rejects unsupported additions. The output can distinguish findings, assumptions, interpretations, and recommendations.

For example, the correct answer may be: “Under the interpretation that both references to the Board denote the same body, the rule is violated because approval precedes the recorded

security review. The document does not establish whether the references denote the same body; therefore the overall result is review required.” A fluent but unsafe renderer might omit the condition and announce a violation. The canonical result prevents this simplification.

The pipeline also supports generation, not only linting. A user supplies an idea or goal. Planning circuits construct a controlled specification with rule-to-plan witnesses. An LLM realizes a draft. Audit circuits analyse the draft using the same evidence and verification path. Revision continues until the canonical audit satisfies the release policy or the budget is exhausted. The generation model never becomes the final oracle for whether its own text satisfies the rules.

Governance surrounds the toolchain. Ontologies, producers, circuits, compilers, formalism contracts, and renderers have versions and digests. Learning agents operate in disposable staging areas. Publication requires benchmarks, semantic mutations, compatibility checks, and explicit approval. Production reads immutable releases. Live model profiles are evaluated separately because deterministic fixtures do not establish provider quality. Current-world knowledge packs have effective dates and jurisdictions.

The architecture should expose a hierarchy of guarantees. Structural guarantees concern exact spans, ordering, and schema validity. Grounding guarantees concern evidence alignment and identity status. Execution guarantees concern consequences of a formal problem. End-to-end guarantees combine them conditionally. The user should be able to see the weakest layer. A result derived from model-proposed events and a mechanically verified transition should not be labelled simply “verified”; it should state that execution is verified over proposed observations.

A small trusted kernel is a major design objective. The kernel includes source canonicalization, schema and type checking, formal compilers, registered executors or proof checkers, replay verifiers, and canonical-result validation. LLM prompts, candidate generators, retrieval heuristics, and explanation models remain outside. Trusted components can still contain bugs, but their behavior is stable, testable, and reviewable. This is a substantial improvement over trusting an evolving foundation model as a monolithic reasoner.

The proposed pipeline is not the only possible implementation, but it captures the obligations revealed by the literature. It explains why a solver is necessary but insufficient. It gives abstract interpretation and symbolic execution distinct roles. It supports multiple formal languages without abandoning a common evidence layer. It permits LLM creativity in discovery, repair, and explanation while preserving symbolic authority over grounding and accepted results. Most importantly, it turns failure into a named artifact—an unsupported type, an unresolved identity, incomplete coverage, a compilation loss, an unsatisfiable formalization, or an unlicensed sentence—rather than a vague hallucination.

7. LongTextJS, CircuitJS, and nllAgent: A Critical Case Study

The public NaturalLanguageLinterAgent repository is one of the few attempts to make the boundary between language interpretation and rule execution explicit in an executable implementation. The project is not a general natural-language reasoner, and its documentation does not present it as one. It is a Node.js 22+ ESM library and command-line runtime with two controlled-language product modes over the same LongTextJS–CircuitJS architecture. Audit mode applies a versioned theory to an existing Markdown document and emits a canonical audit product. Specification mode applies planning theories to a high-level idea and emits a verified generation plan; an optional language model may realize that plan as prose, but the unchanged audit circuits remain the acceptance oracle [AssistOS-AI 2026].

The repository is useful as a case study because it embodies the central thesis of this book in software form: models may propose semantic observations or prose, but production authority is assigned to immutable releases, explicit compatibility checks, restricted circuit execution, verifiers, and canonical artifacts. The project also maintains a public serious-issues register. That practice matters because a large documentation surface can otherwise make architectural intentions look like implemented guarantees. The discussion below distinguishes what is operationally established from what remains a research hypothesis.

No independent peer-reviewed evaluation or third-party technical review of nllAgent, LongTextJS, or CircuitJS was located through searches of the project names, repository name, documentation URL, and combinations with “review,” “evaluation,” “benchmark,” and “replication” as of 2 August 2026. Search results led to the project itself rather than to an external assessment. The judgment in this chapter is therefore the author’s independent architectural review, based on the public repository, documentation, executable examples, declared release process, and current serious-issues file. Absence of an external review is neither praise nor criticism; it means that project claims should remain narrow until independently reproduced.

7.1 What the Public Architecture Implements and Why Its Boundaries Matter

The system defines three production activities that are deliberately separated. Production audit reads an immutable published release, compiles a document into LongTextJS, executes the release's validation circuits, verifies candidate findings, and persists a canonical CNL/Audit-1 result with a rendered report. Production specification compiles an idea into LongTextJS observations, applies planning circuits, and produces a canonical CNL/Plan-1 artifact; optional realization may then generate Markdown and submit it to the same audit machinery. Learning occurs in a disposable staging workspace. A coding agent can generate candidate authoring artifacts and benchmarks, but host-side controls admit only whitelisted files, and a maintainer must explicitly publish and activate a release. This makes learning analogous to governed software development rather than run-time self-modification.

The most important unit of work is not the final report but the run directory. An audit preserves the copied input, source program, selected foundation, compatibility and coverage results, circuit trace, verifier outcomes, findings, canonical machine-readable report, and human-readable view. Planning runs preserve the corresponding plan artifacts. A run that stops because knowledge is missing still records the failure and can create a reusable issue. The design rule that missing knowledge is never reported as compliance is fundamental. It prevents a common failure of document agents: translating "I did not find evidence" into "the document satisfies the rule."

LongTextJS should be understood as a source-side program only in a constrained sense. It does not assign imperative behavior to arbitrary sentences. It compiles a fixed source revision into a finite, queryable world of structures and observations. The valuable properties are source identity, stable spans, versioned observation types, producer identity, coverage, explicit gaps, and replayability. A document-side artifact can be executed because registered consumers can query and transform these relations under declared semantics; it is closer to a semantic intermediate representation or database-loading program than to unrestricted application code.

This distinction is more than terminology. When a model translates a sentence, its output should not become authoritative merely because it is serialized in a programming language. The source text remains evidence; the translated observation records one interpretation of that evidence, generated by a named producer under a schema and model profile. The circuit remains a separate theory. A finding becomes publishable only after a verifier checks the conditions expected by the release. These layers prevent a parser from silently becoming a judge and prevent an explanation from becoming the only record of an inference.

The current foundation pack illustrates the intended semantic scope. Ordinary audit, plan, and benchmark commands select a bounded foundation-core@1.1.0 unless explicitly disabled. The pack recognizes documented controlled-English forms for state, type, exact temporal precedence expressed by "before," exact arithmetic, quantity, and literal emotion assertions. Five replay-verified circuits check only the published logical, temporal, arithmetic, elementary physical, and emotion/type invariants. The selected pack and digest are persisted. Changing political, geographic, social, economic, and legal facts are intentionally excluded from timeless foundation truth. This is a defensible world-model boundary. A system that hardcodes current facts without effective dates, jurisdictions, sources, and conflict policy cannot offer reproducible semantic execution.

The system supports deterministic and model-assisted materialization. Translation selection can prefer a configured AchillesAgentLib language-model agent, use the current constrained coding-agent adapter, or run without a model for deterministic agents. Each model call is given a role-specific skill rather than the entire agent workspace. If a critical observation cannot be materialized within the required schema and resource budget, the result is incompatible, incomplete, or budget exhausted. The model does not control source construction, publication policy, or the meaning of a verifier. This is precisely the correct direction for a federative neuro-symbolic system: the model is powerful at the edge, while symbolic artifacts and host policy control acceptance.

Coverage is treated as an executable obligation rather than an informal confidence score. Circuit compilation works backward from output-reachable nodes to determine which external observations can influence a result. Publication records contracts linking those demands to available producers. Run-time compatibility then asks whether the concrete document actually yielded observations with the required type, status, cardinality, structure, channel, and coverage. A producer's declared capability is not confused with evidence that the current document was successfully processed. When a critical demand is unmet, the run stops at a named boundary.

This demand-driven design is well suited to long documents. It avoids asking a model to produce an all-purpose interpretation of every sentence before any question is known. A circuit can request the semantic dimensions it needs: exact terms, state assertions, quantities, temporal relations, or domain observations. The compiler can materialize only output-relevant observations and persist what was and was not covered. The scientific advantage is not only lower cost. It limits unnecessary semantic commitments and makes incomplete formalization visible.

CircuitJS is the theory-side representation. Agent-editable circuit modules are restricted to one declarative expression, normalized to plain data and passed through the same compiler used for non-code circuit descriptions. They can invoke only registered operators and verifiers; they cannot hide arbitrary executable behavior inside a learned rule. A host application may install one reviewed runtime extension containing real algorithms, but inputs and context are frozen plain-data copies, results must be plain data, cache identity includes the extension digest, and releases that depend on the extension lock that digest. The standard command-line interface does not accept arbitrary executable extension paths.

This boundary addresses one of the most serious practical risks in "agent-learned" rule systems. If a coding agent may generate arbitrary JavaScript in every circuit, learning a linter becomes remote code generation and the semantic behavior of a release becomes difficult to inspect. Restricting ordinary circuit authoring to registered operations creates a finite operational vocabulary. Specialized algorithms remain possible, but they enter through a separate engineering, review, and versioning path. The architecture therefore distinguishes the evolution of theories from the evolution of the trusted runtime.

The verifier boundary is equally important. Circuit execution can create candidate findings, but a verifier controls whether those candidates enter the canonical audit or plan. In the bounded examples, the verifier can reread source anchors, observations, release metadata, and the policy being applied. It is not enough that an execution node emits an object whose fields look like a warning. The verifier establishes the specific property expected by the product.

This does not automatically prove the semantic interpretation correct, but it stops accidental or malicious circuit output from masquerading as an accepted finding.

Specification mode demonstrates how the same architecture can support controlled generation without granting authority to the writer model. Planning circuits build a canonical plan and require rule-to-plan witnesses for applied rules. Optional realization transforms the plan into prose. The candidate prose then passes through audit mode, and the realization model does not judge its own compliance. This is stronger than prompt-only constrained generation because the authoritative object is the plan and audit trace, not the model's self-reported adherence.

The experimental query-first subset is also significant. Canonical LongTextJS is exposed as a finite read-only query world. Typed queries and evidence-aware table values are lowered into ordinary CircuitJS execution. The compiler checks supported relations, expressions, anti-joins, ordering, and hit policies; publication records query contracts and source maps; benchmarks compare direct reference evaluation with lowered execution. The current production editorial release remains deliberately simpler, while aggregates, ordered patterns, native indexes, and advanced reasoning are handled through direct operators or graphs. This restraint is preferable to claiming a mature universal query language before semantics and optimization are stable.

The repository's publication model strengthens reproducibility. Named agents own separate authority material, schemas, extraction profiles, circuits, benchmark suites, candidates, releases, runs, issues, and feedback. A coding agent can propose candidates but cannot make them active. Production reads an immutable release. This structure makes it possible to reproduce which theory, foundation, producer profile, extension digest, and verifier generated a result. It also allows several coding agents or model families to build competing theories without silently sharing an active ontology or release.

The strongest implemented claim is therefore architectural rather than semantic: nllAgent has a disciplined path from immutable source to versioned observations, from observation demand to compatibility, from restricted circuits to verifier-controlled publication, and from a canonical product to a human-readable report. It has explicit stop states. It does not need to pretend that every document is ontologically compatible. It treats model outputs as role-bounded proposals. These are substantial achievements compared with conventional LLM-agent pipelines, where the prompt, interpretation, reasoning, and explanation often exist only in one transient context.

The architecture does not, however, establish that arbitrary natural language has been grounded correctly. Stable spans prove that evidence exists at a location; they do not prove that an actor, event, exception, or modality was interpreted correctly. A typed observation may still refer to the wrong entity. A complete run may be complete only for the bounded observation contract, not for every relevant meaning a human could infer. The project's own documentation and issue register are appropriately cautious on this point. The value of the implementation is that these limitations have places to be represented rather than being hidden inside a final answer.

7.2 Independent Assessment: Strengths, Current Serious Issues, and the Required Evolution

My overall assessment is positive about the trust boundaries and cautious about the semantic claims. nlIAgent is more than a prompt wrapper, more than a vector-retrieval pipeline, and more than a collection of JSON schemas. Its source-side and theory-side artifacts, immutable publication, restricted authoring boundary, replay verification, coverage checks, and canonical products constitute a credible research substrate. The architecture is especially well aligned with the federative pattern identified in recent neuro-symbolic reviews: the neural component proposes or realizes language, while the symbolic component retains its own identity, semantics, and acceptance criteria [Chatzikyriakidis and Lappin 2026].

The project's present evidence nevertheless supports bounded feasibility, not general executable natural language. The default foundation recognizes a small controlled fragment. The editorial demonstration uses narrow operators and verifiers. Query-first compilation is experimental. Controlled realization is limited. No independent evaluation has established formalization fidelity, identity resolution, transfer across domains, error calibration, or superiority over simpler baselines. The right publication strategy is therefore to state exactly which properties are implemented and to attach every stronger claim to a benchmark and guarantee envelope.

The current serious-issues register identifies six engineering and semantic limitations. The first is bounded scheduler incrementality. The runtime has useful dependency and caching mechanisms, but it does not yet claim a fully general incremental fixpoint engine for arbitrary circuits, recursive relations, alternative worlds, and non-monotone updates. This matters for long documents because a small edit should invalidate only the semantic artifacts and circuit states that depend on it. A naive full replay is correct but expensive; an unsound incremental scheduler is fast but dangerous. The solution requires explicit monotonicity and effect contracts, dependency provenance, epoch separation for retractions, and reference comparisons between incremental and full execution.

The second issue is that interpretation support does not yet flow uniformly through every execution path. A document may contain competing readings, worlds, contexts, or presence conditions, but an operator, cache key, aggregation, or verifier can accidentally collapse them. This is the central obstacle to treating LongTextJS as an abstract semantic state. Every value that depends on an interpretation must carry its context. Joins must combine compatible contexts. Aggregation must say whether it ranges within one world or across alternatives. Cache identity must include the context formula. Operators that cannot preserve alternatives should be declared incompatible rather than silently choosing one.

This issue is not a minor feature gap. The core scientific promise of executable natural language is often that the system can say what is true under all admissible readings, what is possible under at least one, and which result depends on a disputed interpretation. If alternatives are lost in an internal execution route, a polished may/must interface would be misleading. The project should therefore make interpretation preservation a whole-runtime invariant and test it through generated finite worlds, metamorphic properties, and differential replay.

The third issue is conservative compatibility for opaque procedural behavior. Registered extensions and operators are necessary because some useful algorithms are awkward to

express declaratively. But a compiler cannot infer precise semantic demands, effects, monotonicity, resource usage, or witness obligations from opaque JavaScript. Conservative rejection is safer than optimistic acceptance, yet excessive opacity reduces composability and makes demand-driven materialization less effective. Each procedural primitive needs a registry-owned contract declaring input and output types, accepted epistemic statuses, coverage requirements, possible effects, determinism, monotonicity, cache behavior, cost class, and verifier-readable witness. The contract should be more authoritative than comments written by the extension author.

The fourth issue is minimal query planning and temporal indexing. A finite query world can execute correctly without sophisticated optimization, but document-scale use will require join planning, selectivity estimates, source-span and entity indexes, temporal interval indexes, and incremental maintenance. These optimizations are not merely performance details. A planner that rewrites queries must preserve evidence multiplicity, context formulas, ordering, anti-join semantics, and coverage. Native indexes must remain disposable accelerators rather than sources of truth. Differential tests should compare optimized and reference evaluation over generated worlds, especially for absence queries, temporal overlap, alternative interpretations, and duplicate evidence.

The fifth issue is deliberately narrow controlled generation. Exact or checked realization of bounded findings and plans is valuable, but it does not yet amount to verified long-form prose. Long-form generation introduces aggregation, pronouns, ellipsis, rhetorical emphasis, discourse relations, and implications created by sentence order. A realization can contain only licensed atomic statements and still imply an unsupported causal or normative narrative. The next step should be proof-carrying realization rather than a broader unconstrained writer. A canonical result should supply claim identifiers, entity and quantity slots, modality, uncertainty, and permitted discourse relations. The generated text should be re-parsed and compared with those claims. Critical mismatches should reject the text; stylistic material outside the checked fragment should be labelled at a lower assurance level.

The sixth issue is scenario-specific mutation support. Ordinary benchmark accuracy can hide rules that are inert, redundant, or incorrectly scoped. A circuit may pass all curated examples even if an exception is ignored because no case activates it. A serious evaluation should generate typed semantic mutants that change thresholds, comparators, quantifiers, negation, modality, identity links, temporal order, evidence status, coverage predicates, exception scope, rule priority, and verifier routes. A test suite “kills” a mutant when the expected canonical result changes in the required way. Surviving mutants reveal weak tests or semantically redundant clauses. Mutation should cover source programs, circuit theories, compiler lowerings, and realization constraints, not only strings in examples.

These six issues are more informative than a generic “prototype” label because they identify where the current architecture can lose either scale or meaning. They also suggest a coherent evolution path. The first milestone should be a bounded formal semantics for the semantic store and query world. It need not formalize unrestricted English. It should define immutable source worlds, observation identity, epistemic status, coverage, alternatives, transactions, query results, publication, and retraction. Key invariants can be model-checked over finite instances and tested property-wise in the implementation.

The second milestone should be interpretation-aware execution. Presence conditions or factorized contexts should accompany alternative-dependent observations. The scheduler,

query engine, primitive registry, cache, verifier, and canonical products should either preserve those contexts or declare incompatibility. May, must, cannot, and unknown should be computed from explicit sets or abstractions of admissible worlds, not from model confidence. This would create the concrete bridge from the existing architecture to abstract interpretation.

The third milestone should be a formalism registry rather than an indefinitely expanding CircuitJS. CircuitJS is best used as an orchestration, evidence-contract, and publication language. Specialized engines should own specialized semantics: relational and Datalog evaluation for closure and joins; decision tables for policy precedence; SAT, SMT, CSP, or MaxSAT for constraints; automata for ordered patterns; temporal solvers and model checkers for traces; symbolic execution for path witnesses; dataflow frameworks for scoped propagation; theorem provers or proof assistants for deductive obligations. Each adapter should declare a typed input fragment, translation assumptions, output certificate, replay checker, and failure modes.

The fourth milestone should be counterexample-guided semantic refinement. A coarse abstract analysis can identify a possible violation. A solver or symbolic executor then seeks a coherent witness. If the witness is satisfiable and replayable, the system reports it with source spans. If it is spurious, an unsatisfiable core or conflict set should identify which identity, temporal, modal, quantity, exception, or coverage distinction was lost. The refinement request should target only the relevant source and producer. The repaired semantic artifact becomes a new version and the counterexample becomes a regression case. This loop would be a genuinely novel synthesis of LLM-based semantic proposal, abstract interpretation, and symbolic execution.

The fifth milestone should be a stronger grounding layer. Type safety and source anchors are necessary but insufficient. The runtime needs candidate identity sets, explicit same/different constraints, temporal validity, unit and jurisdiction checks, source authority, and conflict policies. Model-generated top-one links should not destructively merge objects. Critical entities should use authoritative identifiers or human confirmation. External knowledge should enter through read-only, content-addressed packs with sources, effective intervals, jurisdiction, expiry, conflict resolution, and guarantee ceilings. Web retrieval may propose pack updates, but it should not become implicit timeless truth.

The sixth milestone should be independent evaluation. Deterministic fixtures establish software reproducibility but not the quality of live semantic translation. An opt-in evaluation profile should pin model, provider, prompts, tool versions, temperature, and capture policy; repeat cases to measure variance; record cost and latency; and prohibit silent fallback. A separately implemented reference evaluator should compare query and circuit results where feasible. Human annotation should focus on source-to-semantic fidelity and result-relevant ambiguities. Baselines should include direct LLM judgment, LLM plus retrieval, LLM-to-one-formalism, controlled-language compilation, and deterministic pattern systems. The project should demonstrate when its additional architecture produces measurable value.

A useful assessment can be summarized without inflating the claims. Source preservation, immutable releases, trace persistence, separation of document observations from reusable theory, explicit coverage, restricted circuit authoring, verifier-controlled publication, and governed learning are strong design decisions. General semantic understanding, interpretation-complete execution, production-scale query planning, proof-carrying long-form

generation, comprehensive mutation, and independent empirical validation remain open. The architecture's credibility comes partly from admitting this boundary.

The principal scientific opportunity is not to add more DSL syntax. It is to demonstrate conditional semantic soundness for a bounded but nontrivial fragment. A publishable result could specify a document language containing entities, events, temporal relations, quantities, modality, explicit exceptions, and alternative references; define the admitted interpretation set; prove or mechanically test that every published may/must result is correct relative to that set; show that symbolic refinement eliminates a measurable class of spurious warnings; and verify that generated explanations preserve the canonical result. Such a result would be narrower than "executable English," but substantially more important.

The principal practical opportunity is a new class of document and agent systems that are willing to stop. A normal LLM assistant is optimized to continue producing an answer. A symbolically grounded runtime can return incompatible ontology, unresolved identity, insufficient coverage, unsupported formalism, budget exhaustion, spurious counterexample, or unlicensed realization. These outcomes may appear less impressive in a demonstration, but they are prerequisites for dependable use in science, regulation, engineering, and institutional decision-making.

My independent judgment is therefore that nllAgent has chosen many of the right architectural boundaries and should resist pressure to broaden its claims faster than it broadens its semantic evidence. It is already a credible prototype of governed, source-grounded, theory-separated language analysis. It is not yet evidence that unrestricted natural language can be executed with formal guarantees. Its best path forward is to become an experimental platform for the narrower and more defensible claim developed throughout this book: LLMs discover formal candidates, a symbolic kernel controls grounding and consequence, and natural-language output remains subordinate to a replayable canonical result.

8. Building and Validating a Neuro-Symbolic Runtime

8.1 Implementation Strategy: A Minimal but Scientifically Honest Executable-Language System

The architecture proposed in this book is ambitious only if it is described vaguely. Once its obligations are separated, the first implementable system is comparatively modest. It does not need to understand arbitrary language, construct a universal ontology, or prove every conclusion. It needs to accept a bounded document class, preserve exact evidence, materialize a small family of typed observations, compile a restricted family of rules into one or two formal engines, return witnesses or explicit unknowns, and prevent the final natural-language layer from changing the symbolic result. The scientific value comes from making each limitation visible and measurable.

The first design decision is the target claim. A research prototype should not begin with “the system understands and executes natural language.” It should begin with a claim such as: given documents from a declared genre, a versioned ontology, a finite family of observation producers, and a set of published rules, the runtime determines whether each rule is satisfied, violated, potentially violated, unsupported, or outside the semantic coverage of the release. The result is replayable and linked to source spans. This claim is narrow enough to test and broad enough to support compliance review, requirements analysis, scientific-document linting, protocol inspection, and controlled planning.

The source layer must be deliberately boring. Every document is stored as an immutable revision with a stable digest. Structural units—headings, paragraphs, sentences, list items, table cells, captions, footnotes, and embedded objects—receive persistent identifiers. Normalization is recorded rather than silently applied. Character offsets should refer to a canonical source representation, while mappings preserve positions in the original file. A statement in a table and a visually identical sentence in a paragraph may carry different structural roles; that distinction must survive extraction. When OCR is unavoidable, the OCR output and the page image are separate evidence objects, and uncertainty in the transcription is not collapsed into ordinary text.

This source discipline is not administrative overhead. It is what makes symbolic conclusions contestable. A finding that cites only a generated summary cannot be independently checked. A finding linked to exact text, table cells, and transformation records can be reviewed even if every language model involved has changed. The source package should therefore be considered part of the trusted audit record, although the extractors that create it remain fallible and require format-specific tests.

The second layer is an evidence graph rather than a single parse. The graph contains mentions, entities, events, states, quantities, temporal expressions, modality, relations, and source links. Crucially, it also represents alternatives and status. A model-proposed entity link is not stored as an unconditional equality. It is a candidate identity relation with a producer, score, supporting spans, conflicting evidence, and an epistemic status such as proposed, corroborated, accepted, rejected, or unresolved. A temporal relation inferred from “soon after”

should not be normalized to an arbitrary exact duration; it should retain the qualitative relation and any declared domain interpretation.

This layer should remain finite and task-bounded. The system is not constructing a complete world model. It is materializing the distinctions demanded by the published circuits and by the current query. Query-first materialization, already explored in nlIAgent, is the correct default. A circuit declares that it needs actor identity, document-order relations, deontic modality, and effective dates. The compiler derives those demands before expensive interpretation. Producers then generate only the observations required to decide compatibility and execute the rule. This reduces cost and, more importantly, limits the semantic surface on which errors can occur.

The third layer is a symbolic grounding kernel. “Grounding” is often used loosely to mean that an answer cites a passage. Here it has a stricter meaning: every symbol admitted to formal execution has a typed referent, provenance, scope, and status sufficient for the target formalism. A predicate such as approved(x, y, t) cannot be introduced merely because an LLM produced those tokens. The kernel checks that x and y refer to admissible entity types, that t is a supported time value or interval, that the observation producer is authorized for this predicate, that the source span exists in the declared revision, and that the status meets the rule’s guarantee requirement.

Grounding contracts should distinguish evidence from ontology. Evidence says that a span supports an observation. Ontology says what type of thing the observation denotes and what relations are admissible. Domain axioms say what follows from those relations. World knowledge supplies externally sourced claims. These layers must not be merged. A company registry may establish that two names denote one legal entity during a stated period. A domain ontology may declare that legal entities can sign agreements. A policy theory may declare that a signature by an authorized representative counts as approval. A model may suggest all three connections, but each connection belongs to a different authority and should be accepted or rejected separately.

Identity deserves its own service because it is the most common hidden dependency in long-document reasoning. The service should manage mention clusters, explicit aliases, identifiers, organizational roles, temporal validity, and selected worlds. It should permit several unresolved candidates rather than force early unification. In a first release, identity can be intentionally conservative: explicit identifiers and exact declared aliases are accepted; role-based and contextual links remain proposed; unsupported cross-document identity stops rules that require it. This conservative policy will lower apparent coverage but greatly improve the meaning of accepted results.

Time needs a similarly explicit domain. A practical initial representation can combine exact instants, closed or open intervals, qualitative order, and unknown boundaries. The runtime must distinguish document order from event order and publication time from effective time. Temporal phrases such as “before deployment,” “within thirty days,” “while the authorization remains valid,” and “after any material change” induce different structures. The compiler should reject a temporal rule if the evidence graph has only document-order observations when event-time semantics is required. Silent substitution between temporal domains is a classic source of plausible but invalid conclusions.

Modality and normative force require a small lattice rather than a Boolean. At minimum, the system should distinguish assertion, report, belief, possibility, capability, permission, obligation, prohibition, recommendation, intention, condition, and counterfactual. It should also record the

holder and beneficiary of a norm where relevant. “The provider may disclose data” and “the provider must disclose data” share vocabulary but have opposite compliance consequences. “The document states that the provider complied” is evidence of a report, not necessarily evidence of the underlying event. A formal runtime becomes useful precisely by refusing these collapses.

Once grounded observations exist, the fourth layer computes an abstract semantic state. A minimal domain can represent, for every query-relevant proposition, whether it is true in all admissible interpretations, true in some, false in all, contradicted, or unresolved because of missing coverage. The implementation need not start with a sophisticated lattice library. It can begin with explicit finite alternatives and a mechanically derived may/must summary. As scale grows, abstract domains can summarize identity candidates, numeric ranges, temporal intervals, modal statuses, and provenance sets without enumerating every world.

The abstract state performs two jobs. First, it provides conservative document-wide propagation. If every accepted interpretation links a definition to a term, that relation is a must fact. If only one of three plausible identity assignments creates a conflict of interest, the conflict is a may finding rather than a verified violation. Second, it decides where more expensive execution is justified. A must-safe result can terminate early. A may-violation result triggers a witness search. An unknown caused by absent producer coverage triggers a semantic-compatibility issue rather than a solver call.

The fifth layer is a formalism plan. The plan is not free-form prose written by an agent. It is a typed compilation decision that states what question will be asked of which engine, what observations are inputs, what guarantees the engine returns, and how its result maps to the canonical outcome vocabulary. The choice should be driven by semantic shape rather than fashion.

Relational evaluation is the first engine to implement because many useful document checks reduce to joins, order comparisons, groupings, and set differences. Rules such as “every defined technical term is used consistently,” “every accepted claim has at least one source,” “no requirement identifier is duplicated,” or “all listed risks have a mitigation owner” do not require a theorem prover. A relational core is transparent, efficient, and easy to test. Datalog extends this core when transitive closure or recursive dependencies are required. Examples include inheritance, part-whole propagation, dependency chains, and reachability through citation or requirement graphs.

Decision tables are the second useful formalism for bounded policies. A decision table makes condition combinations and outcomes explicit, exposes overlapping or missing rows, and supports coverage analysis. It is especially appropriate for eligibility, routing, classification, and policy rules whose complexity comes from combinations rather than recursion. A circuit may compile a natural-language policy into a candidate table, but publication should require row-level examples, overlap detection, completeness checks, and human approval of the semantic partition.

SAT, SMT, CSP, and MaxSAT engines become useful when the task contains interacting constraints, arithmetic, schedules, assignments, or optimization. The compiler must choose supported theories and avoid hiding unsupported semantics in uninterpreted strings. An SMT result can prove infeasibility under encoded constraints, produce a satisfying model, or expose an unsatisfiable core. The core should be mapped back to source obligations and assumptions.

This mapping is more informative than asking an LLM to explain a generic solver error; it identifies the smallest formal commitments that cannot jointly hold.

Temporal logic and model checking are appropriate for procedures, workflows, protocols, and lifecycle rules. The evidence graph supplies events and state propositions; the compiler constructs a transition system and temporal property; the checker returns satisfaction, a counterexample trace, or an inconclusive result under bounds. The difficult part is not the temporal checker. It is deciding whether the source defines a complete transition system, whether omitted transitions are impossible or merely undocumented, and whether fairness or environment assumptions are justified. The canonical result must therefore expose the model boundary.

Symbolic execution is appropriate when a rule refers to feasible paths through executable code, a stateful procedure, or a program-like semantic artifact. Its output should be a concrete witness: assignments, branch conditions, state transitions, and source links. In document-only settings, symbolic execution can explore alternative bindings or procedural branches, but it should not be invoked merely because the system contains symbols. The term should retain its path-sensitive meaning. SymGPT is instructive because the natural-language rule is compiled into a violation condition and then tested against actual contract paths rather than against more generated prose [Xia et al. 2026].

Natural logic, finite automata, chart parsing, semiring computation, and specialized theorem provers belong in the portfolio when their structural assumptions fit. Natural logic can efficiently handle monotonicity and certain lexical entailments near surface language. Automata are excellent for ordered local patterns. Chart and semiring methods can maintain multiple parses with shared computation. Proof assistants are appropriate when a small number of high-value claims justify manual formalization and kernel-checked proofs. The architecture should make adding an engine a typed extension, not an invitation to embed arbitrary executable code in circuits.

The sixth layer is counterexample-guided semantic refinement. A may finding is compiled into a feasibility problem. If the solver returns a witness, the runtime replays it against the evidence graph and reports the exact interpretation choices required. If the path is infeasible, the engine returns an unsatisfiable core, failed precondition, or conflict set. The system traces that object back through compilation to the source observations and candidate mappings that produced it. Only then is an LLM asked to help.

The repair prompt should be local and typed. It can ask whether two mentions were incorrectly unified, whether a condition was intended as necessary or sufficient, whether an exception applies, or whether a temporal relation has the wrong endpoint. It should present the relevant source spans, ontology contracts, current alternatives, and the symbolic conflict. The LLM proposes one or more revised semantic artifacts. Those proposals pass through the same grounding and compilation checks as the originals. A human can be inserted when the ambiguity is normatively or scientifically material.

This process is fundamentally different from generic self-reflection. The model is not told that its answer was wrong and invited to improvise another chain of thought. It is given a mechanically identified conflict in a persistent semantic artifact. The repair can be regression-tested. If the repair changes an ontology or producer, publication requires new benchmark cases and a semantic diff. The counterexample becomes a permanent part of the evaluation corpus.

The seventh layer is the canonical result. It should be a stable, solver-independent product that separates verdict, evidence, assumptions, interpretations, execution, and limitations. For an audit rule, it records the rule version, applicable scope, compatibility state, coverage, accepted observations, rejected observations, execution result, witness or proof reference, verifier outcome, and unresolved issues. For a plan, it records goals, constraints, steps, dependencies, resources, alternatives, and the exact checks the plan satisfies. The canonical product is the authority for any subsequent natural-language answer.

The return path to language must be treated as compilation, not free generation. A renderer receives a finite set of licensed claims. Each claim has modality, polarity, quantity, entity references, uncertainty, and provenance. The renderer may choose discourse order and explanatory phrasing, but it may not introduce new entities, strengthen may to must, convert missing evidence into absence, suppress a material assumption, or invent a causal explanation. Sentence-level claim identifiers should remain available for audit. A second checker should compare the rendered text with the canonical result, using deterministic tests for names, quantities, dates, negation, and modality and, where necessary, a separate model constrained to textual entailment rather than open-ended evaluation.

Proof-constrained realization is especially important because users will judge the system by its prose. A perfect symbolic trace can be undermined by an elegant but inaccurate summary. The architecture should therefore make verbosity optional but fidelity mandatory. In high-assurance mode, the system may generate plain, repetitive language because each sentence follows a template licensed by the result type. In explanatory mode, an LLM can improve readability, but unsupported sentences are rejected and regenerated. The canonical artifact, not the prose, remains the record of decision.

Security and governance must be integrated from the beginning. Natural-language documents are untrusted inputs and can contain prompt injection, adversarial examples, misleading formatting, or code fragments. Semantic producers should receive source data through typed interfaces rather than a privileged system prompt that can be overwritten by document content. Generated formal artifacts must be parsed by restrictive grammars. Solvers should run under resource limits. Runtime extensions should be reviewed, versioned, and digest-locked. Learning workspaces should not publish directly to production. These boundaries are already partly present in nllAgent and should be strengthened rather than bypassed for convenience [AssistOS-AI 2026].

A realistic implementation programme can proceed in five releases. The first release should support immutable Markdown or DOCX ingestion, source spans, structural observations, a relational query engine, restricted controlled-English observations, coverage states, reusable circuits, replay verification, and canonical findings. It should solve bounded tasks such as provenance completeness, definition consistency, requirement coverage, ordered section rules, and simple modal distinctions. No general identity, temporal inference, or current-world retrieval should be claimed.

The second release should add typed identity and time. Identity begins with explicit identifiers, aliases, and document-declared roles; temporal semantics begins with exact dates, intervals, document order, and a small set of event-order patterns. Benchmarks should contain ambiguous aliases, changing roles, overlapping intervals, and deliberately missing links. The system should stop rather than guess when a circuit requires stronger identity than the release provides.

The third release should add decision tables and SMT/CSP execution, with unsatisfiable-core mapping and counterexample-guided semantic refinement. This release can address eligibility policies, schedules, assignment constraints, quantitative consistency, and bounded procedure feasibility. Every solver result must be replayed through an independent verifier or a smaller checker where possible. Mutation testing should alter constraints and observations to demonstrate that benchmark suites detect semantic regressions.

The fourth release should add a temporal/model-checking adapter and a path-sensitive symbolic-execution adapter for one concrete domain, such as workflow specifications or smart-contract rules. The purpose is not broad coverage but proof that the formalism registry and canonical certificate interface support genuinely different semantics. Cross-formalism consistency tests should check that shared grounded facts retain identity, time, modality, and provenance across lowerings.

The fifth release should add planning and controlled realization. An idea is compiled into a canonical plan, checked against formal constraints, realized into prose, and audited by the same rule system. The evaluation must distinguish plan validity, realization fidelity, linguistic quality, and repair efficiency. The plan does not become trustworthy merely because prose was generated from a structured object; it becomes more auditable because the object and its checks are persistent.

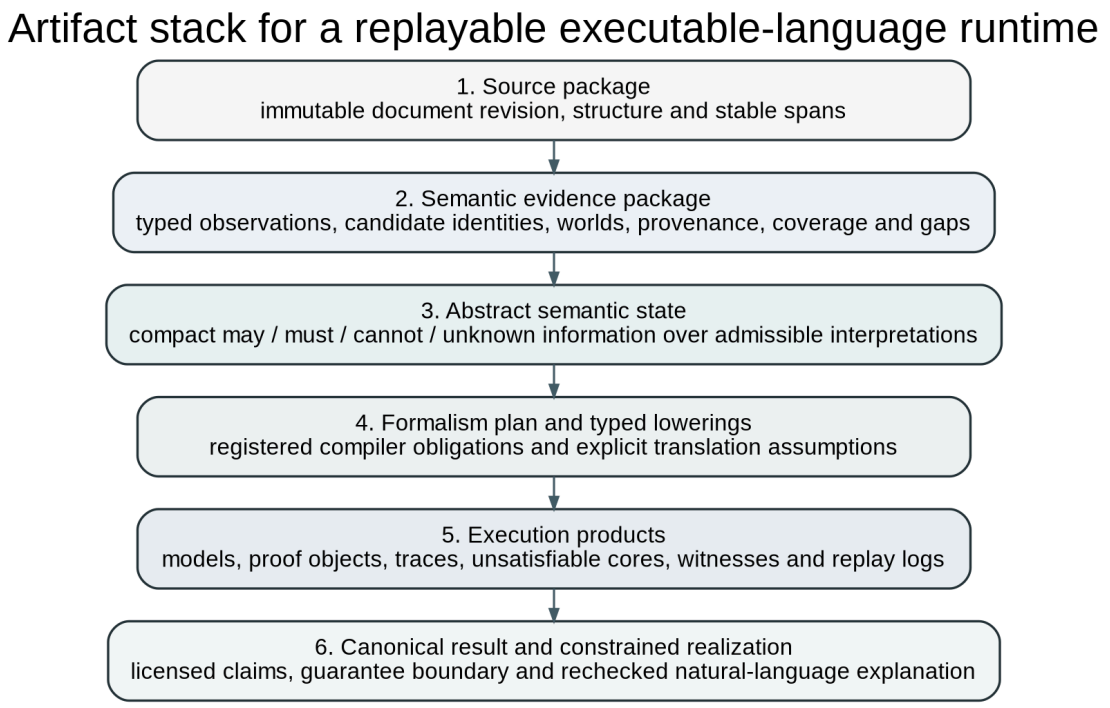


Figure 4. Artifact stack for a replayable executable-natural-language runtime. Each layer is versioned, source-mapped, and independently inspectable.

This staged implementation embodies an important methodological principle: semantic ambition should grow only when the system can detect that its previous representation is inadequate. The runtime should not hide unsupported language behind model confidence. It should expose compatibility gaps as first-class outcomes and turn recurring gaps into new producer, ontology, engine, and benchmark work. Executable natural language advances through governed expansion of a soundness envelope, not through one leap to universal understanding.

8.2 Research Questions, Benchmarks, and a Publication Programme

The architecture becomes a scientific contribution only when its claims are refutable. A demonstration in which an LLM generates a plausible formal artifact and a solver returns an answer is insufficient. The same model can generate the benchmark, the translation, and the explanation, creating a closed loop of correlated error. A serious evaluation must decompose the pipeline and test each obligation independently. It must also include failure conditions under which the proposed approach should be rejected or narrowed.

The first research question concerns representation: does maintaining several admissible semantic interpretations improve reliability over selecting one translation? The corresponding hypothesis is that candidate-set semantics, combined with may/must analysis, reduces false assurance on ambiguous and underspecified inputs. It may lower apparent answer rate because more cases become conditional or unknown. That is not necessarily a defect. The relevant comparison measures whether accepted must conclusions are more often correct and whether may findings capture materially plausible alternatives without overwhelming reviewers.

A benchmark for this question should contain texts with expert-annotated interpretation sets rather than one gold logical form. Minimal pairs should vary quantifier scope, negation, exception attachment, coordination, coreference, temporal boundaries, modal force, and jurisdiction. Some items should have one clearly intended reading, some several legitimate readings, and some insufficient evidence for any determinate answer. Systems are evaluated on coverage of admissible readings, exclusion of inadmissible readings, calibration of may/must status, and downstream verdicts. Exact graph match is too brittle; semantic equivalence should be tested through query suites or bidirectional entailment under the benchmark ontology.

The second research question concerns refinement: are solver-derived conflicts more useful for repairing formalization than generic model self-critique? The hypothesis is that counterexamples, unsatisfiable cores, and failed proof obligations produce more localized, stable, and sample-efficient repairs because they identify the dependency responsible for the result. A controlled experiment should compare four conditions: direct retry, verbal self-reflection, execution-error feedback, and counterexample-guided semantic refinement with source mapping. The same model, temperature, and budget should be used where possible.

Repair quality should be measured not only by whether the target example becomes correct. The repair must preserve unrelated cases, reduce the semantic diff, and avoid introducing unsupported assumptions. Mutation-based regression suites are essential. A repair that solves one example by weakening the rule globally is not a success. The experiment should record number of iterations, tokens, human interventions, surviving mutants, changed observations, and whether the final artifact was independently replayable.

The third research question concerns the evidence contract. Does a typed separation of source spans, observations, ontology assertions, domain axioms, and world-knowledge premises improve transfer and auditability compared with a flat knowledge graph or free-form chain of thought? The hypothesis is that typed contracts reduce silent premise introduction and make cross-domain reuse more predictable. The likely cost is authoring overhead and more stopped runs.

The benchmark should include the same rule applied across documents from several domains, with controlled variation in vocabulary and ontology. For example, a general

requirement that every accepted claim have traceable evidence can be tested on research papers, policy documents, technical specifications, and plans. Structural observations may transfer directly; domain-specific entity and relation producers may not. The evaluation records which compatibility checks stop, whether the stop reason is correct, and how much new semantic material is required to resume execution. Successful transfer is not merely obtaining an answer; it is accurately recognizing what can and cannot be reused.

The fourth research question concerns formalism choice. Does routing to a portfolio of specialized engines outperform a universal intermediate language under equal semantic supervision? Beiser and Penz show that intermediate-language choice can change overall and execution accuracy substantially on neuro-symbolic reasoning tasks [Beiser and Penz 2025]. The hypothesis here is more specific: a typed router that selects the simplest adequate formalism will improve compilation success, explanation quality, and efficiency without sacrificing cross-task consistency.

A meaningful comparison should not allow the portfolio system more hidden information. Each task receives the same source evidence and ontology. The single-formalism baseline might use first-order logic, ASP, SMT, or a general graph query language. The portfolio system may choose relational queries, Datalog, decision tables, SMT, temporal logic, or symbolic execution. Metrics include formalization validity, semantic equivalence, solver success, runtime, witness quality, and maintenance cost. Cross-formalism cases should require two engines and test whether shared symbols remain consistent.

The experiment should also seek counterevidence. A universal representation may outperform the portfolio in narrow domains because routing introduces errors and multiple compilers increase the trusted base. The expected conclusion is not that portfolios are always superior. It is that formalism plurality is justified only when the task distribution genuinely contains different computational structures and when interfaces preserve semantics.

The fifth research question concerns the final language layer. Does proof-constrained realization reduce unsupported or semantically strengthened claims while preserving acceptable readability? The hypothesis is that a canonical result plus claim-licensed generation substantially lowers hallucination and modality drift relative to direct answer generation or answer generation from a raw solver trace. The cost may be less natural prose and more regeneration.

The benchmark should include canonical results with subtle uncertainty, conflicting evidence, nested conditions, quantities, and dates. Human evaluators judge factual fidelity, completeness of material limitations, readability, and usefulness. Deterministic checkers measure entity, number, date, polarity, and modality preservation. An adversarial set should tempt the renderer to produce a stronger, simpler conclusion than the result supports. The central metric is not stylistic preference but the rate at which the prose exceeds the authority of the symbolic product.

The sixth research question concerns federative architecture itself. The 2026 systematic review of neuro-symbolic NLP reports promising gains for federative systems but warns that the evidence is confounded by task differences [Chatzikyriakidis and Lappin 2026]. A direct experiment should compare a federative pipeline with an injective or end-to-end learned alternative on the same tasks, data, and model scale. The relevant benefits may appear less in raw accuracy than in auditability, abstention, error localization, and repair.

The study should therefore report several dimensions. Raw task accuracy remains necessary. It should be accompanied by semantic-form accuracy, solver-conditioned accuracy, unknown calibration, reproducibility across model versions, explanation fidelity, human review time, and the fraction of errors attributable to extraction, grounding, compilation, execution, or realization. A modular system that matches rather than beats end-to-end accuracy may still be preferable for high-assurance work if it turns silent errors into diagnosable artifacts. Conversely, modularity should not be defended as an intrinsic virtue if it merely adds bureaucracy without improving evidence.

A seventh research question concerns abstract interpretation over semantic alternatives. Can a useful abstract domain be defined that is conservative relative to a benchmark interpretation set and more scalable than explicit enumeration? The first experiments can use finite sets as the concrete reference. Abstract domains for identity, temporal intervals, modality, quantities, and coverage are implemented separately and then combined. Soundness is checked mechanically against enumeration on small cases. Precision is measured by the rate of may findings that can be eliminated by witness search or refinement.

This programme should be careful with the word soundness. If expert annotation defines a finite interpretation set, the abstract interpreter can be sound relative to that set. It cannot thereby prove that the set exhausts human meaning. The contribution is an executable conditional semantics and evidence about its usefulness. A later theoretical paper can define the admissibility relation, abstraction and concretization functions, transfer conditions, and refinement operators. The empirical system should not wait for a complete semantics of language, but it should make the conditional nature of its claims explicit.

An eighth research question concerns symbolic execution over semantic programs. Does path-sensitive witness construction improve reviewer understanding and defect discovery relative to static rule outcomes? In a workflow or policy domain, the runtime can compare a simple rule evaluator with symbolic path exploration. The evaluator says that a violation is possible; the executor returns a concrete sequence of events and assignments. Human reviewers judge whether the witness is intelligible and useful. Ground-truth defects and seeded mutants determine whether the witness corresponds to a real case.

Path explosion must be part of the evaluation rather than treated as an implementation inconvenience. The study should vary branching, loop bounds, identity alternatives, and temporal choices. LLM guidance may prioritize paths, as recent LLM-guided symbolic-execution work suggests, but the engine must distinguish heuristic coverage from formal completeness. Coverage metrics should report explored states, pruned states, solver timeouts, and unsupported external behavior. A witness is strong evidence for existence; failure to find one is not evidence of absence unless the exploration bound and model justify that conclusion.

The benchmark architecture should be layered. The source layer tests ingestion and anchoring. The observation layer tests extraction and status. The grounding layer tests identity, time, modality, quantities, and authority. The formalization layer tests semantic equivalence and target-language validity. The execution layer tests solver outputs and certificates. The result layer tests verdict composition and uncertainty. The realization layer tests fidelity of natural-language answers. Each example should identify the earliest layer at which an injected fault ought to be detected.

Benchmark cases should be small enough to validate manually and composable into longer documents. Small examples provide trustworthy ground truth for subtle logic. Long composites

test retrieval, indexing, identity continuity, and scalability. The same underlying rule can appear in several linguistic forms and document structures. Paraphrase families test invariance; contrast sets change one semantic feature and test sensitivity. The benchmark must avoid the common failure in which a system performs well by memorizing surface patterns while ignoring the intended distinction.

Mutation testing should be central. Mutations can change a quantifier, negate a condition, move an exception, alter a date boundary, merge identities, delete evidence, change modal force, swap actor and object, duplicate a requirement, or modify a circuit threshold. A robust suite should detect the expected change and leave unrelated results stable. Mutation score is not a complete quality measure, but it reveals tests that merely confirm one happy path. For nllAgent, executable mutations would directly address one of the repository's acknowledged gaps [AssistOS-AI 2026].

The evaluation should include prompt injection and semantic adversaries. Documents can contain instructions directed at the model, misleading headings, quoted policy text, adversarial Unicode, hidden content, or statements that resemble ontology declarations. The source parser must preserve these as data. Semantic producers should operate under fixed system authority. The benchmark should test whether document text can alter the circuit, suppress findings, or cause privileged tool calls. Security outcomes should be recorded separately from semantic accuracy.

Live-model evaluation must be reproducible enough to support scientific claims. Every run should pin provider, model identifier, request schema, temperature, seed when supported, context construction, and retry policy. Inputs and outputs should be captured subject to privacy constraints. Repeated trials estimate variance. Deterministic doubles remain useful for software tests but cannot establish translation quality. At least two model families should be tested where feasible to distinguish architectural effects from one provider's behavior.

Human evaluation is necessary but should be narrowly designed. Experts should not be asked whether an answer "looks good." They should judge specific obligations: whether the formalization preserves the source, whether identities are justified, whether a witness is feasible, whether the explanation states all material assumptions, and whether an unknown is appropriately returned. Inter-rater disagreement is itself evidence of semantic ambiguity and should not be forced into a single gold label without analysis.

The core quantitative metrics should include source-anchor accuracy; observation precision and recall by status; ontology and type validity; identity-link precision, recall, and calibration; temporal and modal relation accuracy; candidate-set coverage; may/must soundness relative to annotated interpretations; formalization syntax and semantic equivalence; solver success; witness validity; proof-check success; false assurance rate; appropriate abstention; coverage; latency; cost; review time; mutation score; and realization fidelity. A single aggregate score would hide the mechanism and should be secondary.

False assurance deserves special emphasis. It measures cases in which the system presents a determinate verified conclusion although the source, grounding, or interpretation set does not justify it. In high-assurance applications, reducing false assurance may matter more than maximizing answered questions. The system should receive credit for a precise stopped state when a direct LLM confidently invents a bridge. Appropriate unknown is a positive result.

Baselines should include direct LLM answering, chain-of-thought or equivalent reasoning prompting, tool-augmented LLM execution without persistent formal artifacts, single-formalism translation, candidate-set translation without abstract interpretation, symbolic execution without semantic refinement, and the complete architecture. For the nllAgent case study, the current bounded foundation release can serve as an implemented baseline, while proposed identity, abstract-semantics, and multi-formalism extensions are evaluated incrementally. This avoids comparing a full research vision only against weaker external systems.

The publication programme can be organized around five contributions rather than one oversized claim. The first paper should be a position and architecture paper that defines executable natural language, the trust boundary, the formalization gap, the soundness envelope, and the multi-formalism design. Its contribution is conceptual clarity and a reference architecture, supported by a systematic survey and a small working demonstrator.

The second paper should formalize the provenance-bearing semantic pivot and conditional soundness. It defines source objects, typed observations, candidate interpretations, may/must semantics, and compatibility. It proves soundness of the abstract computation relative to a finite admissible interpretation set and reports experiments on identity, time, modality, and coverage. This is the programming-languages and formal-semantics core.

The third paper should introduce counterexample-guided semantic refinement. It defines mappings from solver conflicts to source-linked semantic commitments and compares refinement with self-reflection and generic execution feedback. The strongest result would be lower false assurance, smaller repairs, better regression preservation, and reduced human effort. Even a negative result would be valuable if it identifies which solver feedback is too low-level to guide semantic repair.

The fourth paper should present the formalism portfolio and typed engine contracts. It evaluates relational, Datalog, decision-table, SMT, temporal, and symbolic-execution adapters on a common artifact model. The paper should emphasize semantic preservation and certificates rather than the number of integrations. A small set of genuinely different engines is more persuasive than many shallow wrappers.

The fifth paper should be the nllAgent system and evaluation paper. It should clearly distinguish the implemented release from the broader runtime. It can document immutable source revisions, LongTextJS, CircuitJS, demand-driven observations, compatibility, coverage, restricted execution, learning and publication governance, replay verification, canonical audit and plan products, and the serious-issues process. The evaluation should include executable mutation, live-model records, identity cases, current-world pack governance, planning benchmarks, and an independent or separately implemented verifier where feasible.

A sixth later paper could study proof-constrained realization. It would compare direct generation, template rendering, constrained LLM rendering, and post-hoc verification. The central contribution would be a claim-licensing representation and evidence that readable language can be generated without changing modality, uncertainty, or factual content. This work connects formal execution to the practical need for accessible human communication.

The project should define explicit failure criteria. The candidate-set approach should be reconsidered if it produces so many alternatives that may/must results are uninformative and refinement cannot recover precision. The formalism portfolio should be narrowed if routing and cross-formalism bugs exceed the benefits of specialization. LLM-assisted formalization should

be restricted to authoring support if semantic equivalence remains poor under realistic documents. Proof-constrained realization should fall back to templates if model-generated prose repeatedly exceeds its license. Current-world reasoning should remain outside the trusted core if source, time, jurisdiction, and conflict controls cannot be enforced.

Scientific honesty also requires comparison with simpler alternatives. Many document checks can be solved by deterministic parsers, regular expressions, schema validation, database queries, or human-authored rules. An LLM is justified when language variation, ontology mapping, or semantic proposal materially reduces authoring cost or increases coverage. A symbolic core is justified when exact state, replay, witnesses, or independent verification matter. A research paper should report when a simpler system performs equally well. Executable natural language is not a reason to formalize every editorial preference or insert a solver into tasks that need only a reliable parser.

The likely practical outcome is not a machine that executes all natural language. It is a family of governed semantic compilers for document and agent domains, sharing source, grounding, execution, and certificate infrastructure. Each compiler has a declared semantic envelope. LLMs make those compilers easier to author, adapt, and explain. Formal methods make their accepted results reproducible and contestable. The combination is useful precisely because neither component is asked to provide what it cannot reliably provide alone.

The research programme is therefore both more modest and more consequential than the slogan “natural language as code.” Natural language remains an open, social, context-dependent medium. The runtime creates executable projections of it for declared questions. Those projections can be rigorous enough to support real audits, plans, protocols, and scientific checks, provided the system records the distance between source and formal object instead of pretending that the distance has disappeared.

Conclusion: From Probabilistic Form to Symbolic Consequence

The recent revival of neuro-symbolic AI has produced a recognizable architectural pattern. Large language models are used where the search space is linguistic, heterogeneous, or difficult to enumerate. Formal systems are used where exact state, consistency, exhaustive bounded search, witnesses, or proofs matter. The interesting question is no longer whether neural and symbolic methods can be combined. They plainly can. The important question is where authority lies and what is independently checked.

The survey shows that the most credible systems do not grant the language model final authority over the result. Controlled languages translate a restricted surface into formal semantics. Semantic parsers generate executable queries or programs. PAL delegates computation to Python. SatLM delegates constraints to a solver. Logic-LM and LINC delegate deduction to logical engines. NL2TL and nl2spec translate requirements to temporal specifications. Planning systems use formal models and verification tools to maintain long-horizon consistency. SymGPT compiles natural-language ERC rules into constraints and uses symbolic execution to construct concrete contract paths. SAIL uses LLMs to search for abstract transformers but admits only candidates that satisfy mechanical obligations. These systems differ greatly, yet they converge on the principle that generation proposes and an independent mechanism decides.

This convergence should not be overstated. Solver use does not verify understanding. The formalization gap remains the central problem. A theorem prover can derive a valid conclusion from predicates that misrepresent the text. A program can return the expected answer through a spurious path. An SMT model can satisfy constraints that omit the user's decisive requirement. A symbolic executor can expose a real software path for a mistranslated rule. The strongest current systems reduce deductive and computational error while relocating risk to semantic parsing, grounding, ontology, and scope.

The appropriate response is not to abandon formal tools because translation is imperfect. It is to make translation a persistent, typed, reviewable artifact. The source must remain available. Every semantic observation must identify its producer, evidence, status, and ontology. Identity, time, modality, quantities, and authority must be represented rather than inferred afresh during each answer. Alternative interpretations must remain alternatives when the distinction affects the result. A symbolic engine should execute only grounded objects that meet explicit contracts.

Abstract interpretation provides a useful theory for the global side of this architecture. It asks what can be concluded conservatively from a large set of possible states. Applied to natural language, the concrete states are not undefined "meanings" but bounded semantic structures admissible under a declared grammar, ontology, source revision, and grounding policy. An abstract state can then represent what must hold, may hold, cannot hold, conflicts, or remains unknown. The guarantee is conditional on the interpretation set, but conditional soundness is still meaningful when the conditions are explicit and tested.

Symbolic execution provides the local complement. When an abstract analysis reports a possible violation, symbolic execution or another constraint engine can search for a concrete interpretation and path. A successful search returns a witness that a reviewer can inspect. A failed search exposes an overly coarse abstraction or an incompatible semantic choice.

Mapping that failure back to source spans creates counterexample-guided semantic refinement: the model or human revises the smallest relevant formal commitment, and the analysis is replayed. This is a more disciplined use of LLM feedback than generic self-critique because the error signal is tied to a formal dependency.

No single target language is likely to support every useful form of executable prose. The literature already demonstrates the dependence on intermediate-language choice, and contemporary systems succeed by using Python, SQL, first-order logic, temporal logic, PDDL, constraint languages, or domain-specific grammars according to the task. A practical runtime should therefore use a provenance-bearing semantic pivot and a typed portfolio of formal engines. Relational and Datalog evaluation handle document queries and closure; decision tables handle finite policies; SAT/SMT/CSP handle interacting constraints; model checking handles temporal behaviors; symbolic execution handles path witnesses; automata handle ordered patterns; natural logic handles selected entailments; proof assistants handle high-value formal claims.

Plurality introduces its own engineering risks. Every compiler can distort semantics, every engine expands the trusted base, and routing can fail. The portfolio is justified only if interfaces are typed, shared symbols retain identity and provenance, engine guarantees are explicit, and results are normalized into a canonical certificate-bearing product. A universal logic may remain preferable in a narrow domain. The architectural claim is not that plurality is always better; it is that broad executable language requires different computational structures and should not disguise them behind one vague DSL.

The return from symbols to language is part of the trust boundary. A final LLM must not be permitted to turn a conditional result into an unconditional recommendation, a possible violation into a confirmed breach, or missing evidence into evidence of absence. The canonical result should define a finite set of licensed claims with modality, quantities, entities, assumptions, and provenance. The renderer can improve readability but should remain subordinate to that authority. Proof-constrained realization is the natural completion of the architecture.

The public nllAgent implementation is significant because it already embodies several of these disciplines. LongTextJS preserves a source-grounded document program; CircuitJS represents reusable theories; observation demand is derived from circuits; compatibility and coverage are explicit; restricted modules and reviewed extensions limit hidden execution; learning takes place in disposable staging; publication is manual and immutable; replay verification controls canonical findings; and missing knowledge is not treated as compliance. These decisions make the project a credible platform for research even though they do not establish general semantic understanding.

The same public materials identify the important unfinished work: general identity materialization, complete typed primitive contracts, executable semantic mutation, independent shadow evaluation, live-model benchmarking, planning and realization benchmarks, and governed current-world knowledge packs. No independent peer-reviewed assessment of nllAgent was located for this edition. The judgment offered here is therefore an architectural one: the implementation has chosen unusually sound boundaries, but its strongest defensible claims remain bounded structural and controlled-language analysis. Its value will increase if semantic breadth is expanded only through explicit producers, contracts, benchmarks, and evidence.

The proposed next step is not to replace LongTextJS or CircuitJS with a larger universal language. LongTextJS should mature as a finite, queryable, provenance-bearing semantic world. CircuitJS should remain a restricted orchestration and evidence-contract layer. Specialized registered engines should perform relational, constraint, temporal, dataflow, automata, or path-sensitive execution. May/must semantics should become operational. Counterexamples should drive focused semantic refinement. The canonical CNL products should control the final language output.

A rigorous research programme can test each part. Candidate-set semantics can be compared with single-translation systems. Solver-derived refinement can be compared with self-reflection. Typed evidence contracts can be tested for transfer and false assurance. A formalism portfolio can be compared with a universal intermediate language. Abstract domains can be checked against explicit interpretation sets. Symbolic witnesses can be evaluated for defect discovery and reviewer usefulness. Proof-constrained realization can be tested for fidelity. Modular and end-to-end architectures can be compared on the same tasks rather than on different benchmark families.

The decisive metric should be false assurance, not merely answer rate. A system that returns “unknown because identity is unresolved” may be more valuable than one that produces a confident verdict by silently merging two people. A system that stops because current-world evidence is stale is more trustworthy than one that retrieves an unsupported fact. A system that exposes three interpretations is better than one that chooses the most fluent. Appropriate abstention is part of execution semantics.

Executable natural language should therefore be understood as a governed compilation relation, not an intrinsic property of prose. The source text remains richer than any executable projection. The system selects a task, materializes relevant semantics, preserves uncertainty, grounds symbols, compiles to an appropriate formalism, executes under explicit bounds, verifies the result, and returns a constrained explanation. What becomes executable is not language in its entirety but a declared semantic projection whose assumptions and limits remain visible.

This narrower conception is sufficient for substantial practical impact. Scientific claims can be checked for provenance and consistency. Requirements can be linked to tests and lifecycle states. Policies can be compiled into decision structures. Plans can be verified before realization. Protocol rules can be mapped to state and path analyses. Long documents can be linted by reusable theories without asking a model to remember every detail. Agents can use natural language as an authoring and coordination layer while delegating exact consequence to symbolic runtimes.

The central principle can be stated without formal symbolism: let the LLM discover possible form; let the grounding kernel decide what the symbols are allowed to denote; let formal engines determine what follows; and let the final LLM explain only what the symbolic record licenses. This principle does not solve every problem of language. It creates a defensible boundary between interpretation and consequence, which is the necessary foundation for natural language to participate safely in executable systems.

References

Abzianidze, Lasha, Rik van Noord, Hessel Haagsma, and Johan Bos. 2019. “The First Shared Task on Discourse Representation Structure Parsing and Evaluation.” Proceedings of the IWCS Shared Task on Semantic Parsing. Association for Computational Linguistics. https://aclanthology.org/W19-1201/

Abzianidze, Lasha. 2017. “LangPro: Natural Language Theorem Prover.” Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Association for Computational Linguistics. https://aclanthology.org/D17-2022/

Angeli, Gabor, and Christopher D. Manning. 2014. “NaturalLI: Natural Logic Inference for Common Sense Reasoning.” Proceedings of EMNLP 2014. Association for Computational Linguistics. https://aclanthology.org/D14-1209/

AssistOS-AI. 2026. “NaturalLanguageLinterAgent (nlIAgent): LongTextJS–CircuitJS Architecture, Documentation, and Serious Issues.” Public repository and generated documentation, inspected 2 August 2026. https://github.com/assistos-ai/nlIAgent

Badreddine, Samy, Artur d’Avila Garcez, Luciano Serafini, and Michael Spranger. 2022. “Logic Tensor Networks.” Artificial Intelligence 303: 103649. https://doi.org/10.1016/j.artint.2021.103649

Baldoni, Roberto, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. 2018. “A Survey of Symbolic Execution Techniques.” ACM Computing Surveys 51(3): 1–39. https://doi.org/10.1145/3182657

Beg, Arshad, Diarmuid P. O’Donoghue, and Rosemary Monahan. 2025. “A Short Survey on Formalising Software Requirements Using Large Language Models.” arXiv:2506.11874. https://doi.org/10.48550/arXiv.2506.11874

Beiser, Alexander, and David Penz. 2025. “Making LLMs Reason? The Intermediate Language Problem in Neurosymbolic Approaches.” arXiv:2502.17216. https://arxiv.org/abs/2502.17216

Bos, Johan. 2008. “Wide-Coverage Semantic Analysis with Boxer.” Proceedings of the 2008 Conference on Semantics in Text Processing. https://aclanthology.org/W08-2222/

Chatzikyriakidis, Stergios, and Shalom Lappin. 2026. “Neuro-Symbolic NLP: Taxonomy, Assessment, and Directions.” Frontiers in Artificial Intelligence 9. https://doi.org/10.3389/frai.2026.1797587

Chen, Lingfeng, Tao Xiao, Masanari Kondo, and Yasutaka Kamei. 2026. “Directed Symbolic Execution for Vulnerability Discovery: An LLM-Guided Approach in KLEE.” arXiv:2607.21676. https://arxiv.org/abs/2607.21676

Chen, Yongchao, Rujul Gandhi, Yang Zhang, and Chuchu Fan. 2023. “NL2TL: Transforming Natural Languages to Temporal Logics Using Large Language Models.” Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics. https://aclanthology.org/2023.emnlp-main.985/

Clarke, Edmund, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. “Counterexample-Guided Abstraction Refinement.” Computer Aided Verification, LNCS 1855, 154–169. https://doi.org/10.1007/10722167_15

Cosler, Matthias, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. “nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models.” Computer Aided Verification, LNCS 13964, 383–396. https://doi.org/10.1007/978-3-031-37703-7_18

Cousot, Patrick, and Radhia Cousot. 1977. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” Proceedings of POPL 1977, 238–252. https://doi.org/10.1145/512950.512973

Delvecchio, Giovanni Pio, Lorenzo Molfetta, and Gianluca Moro. 2025. “Neuro-Symbolic Artificial Intelligence: A Task-Directed Survey in the Black-Box Models Era.” Proceedings of IJCAI 2025, Survey Track, 10418–10426. https://doi.org/10.24963/ijcai.2025/1157

Dong, Wenkai, and Yifan Wang. 2026. “From Interpretation to Compilation: Compilation-Based Execution of Semantic Operators.” arXiv:2607.13407. https://doi.org/10.48550/arXiv.2607.13407

Ferrari, Alessio, and Paola Spoletini. 2025. “Formal Requirements Engineering and Large Language Models: A Two-Way Roadmap.” Information and Software Technology 181: 107697. https://doi.org/10.1016/j.infsof.2025.107697

Fuchs, Norbert E., and Rolf Schwitter. 1996. “Attempto Controlled English (ACE).” Proceedings of the First International Workshop on Controlled Language Applications. https://attempto.ifi.uzh.ch/site/pubs/papers/ace96.pdf

Fuchs, Norbert E., Kaarel Kaljurand, and Tobias Kuhn. 2008. “Attempto Controlled English for Knowledge Representation.” Reasoning Web, LNCS 5224, 104–124. https://doi.org/10.1007/978-3-540-85658-0_3

Gao, Luyu, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. “PAL: Program-Aided Language Models.” arXiv:2211.10435. https://arxiv.org/abs/2211.10435

Goldman, Omer, Veronica Latcinnik, Udi Naveh, Amir Globerson, and Jonathan Berant. 2018. “Weakly Supervised Semantic Parsing with Abstract Examples.” Proceedings of ACL 2018. Association for Computational Linguistics. https://aclanthology.org/P18-1168/

Grimaldi, Pasquale. 2026. “pamaldi at SemEval-2026 Task 11: Neuro-Symbolic Syllogistic Reasoning via LLM-Guided Structure Extraction and Deterministic Validation.” Proceedings of the 20th International Workshop on Semantic Evaluation. Association for Computational Linguistics. https://doi.org/10.18653/v1/2026.semeval-1.186

Gu, Qiuhan, Avaljot Singh, and Gagandeep Singh. 2026. “SAIL: Sound Abstract Interpreters with LLMs.” Proceedings of the ACM on Programming Languages 10 (PLDI): 1534–1559. https://doi.org/10.1145/3808308

Guu, Kelvin, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. “From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood.” Proceedings of ACL 2017. Association for Computational Linguistics. https://aclanthology.org/P17-1016/

Hamilton, Kyle, Aparna Nayak, Bojan Božić, and Luca Longo. 2024. “Is Neuro-Symbolic AI Meeting Its Promises in Natural Language Processing? A Structured Review.” Semantic Web 15(4): 1265–1306. https://doi.org/10.3233/SW-223228

Han, Simeng, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, Lucy Sun, Alex Wardle-Solano, Hannah Szabó, Ekaterina Zubova, Matthew Burtell, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Alexander Fabbri, Wojciech Kryściński, Semih Yavuz, Ye Liu, Xi Victoria Lin, Shafiq Joty, Yingbo Zhou, Caiming Xiong, Rex Ying, and Dragomir Radev. 2024. “FOLIO: Natural Language Reasoning with First-Order Logic.” Proceedings of EMNLP 2024. Association for Computational Linguistics. https://aclanthology.org/2024.emnlp-main.1229/

Hao, Yilun, Yongchao Chen, Yang Zhang, and Chuchu Fan. 2025. “Large Language Models Can Solve Real-World Planning Rigorously with Formal Verification Tools.” Proceedings of NAACL 2025, 3434–3483. https://aclanthology.org/2025.naacl-long.176/

Hu, Ruikang, Shaoyu Lin, Yeliang Xiu, and Yongmei Liu. 2025. “LTRAG: Enhancing Autoformalization and Self-refinement for Logical Reasoning with Thought-Guided RAG.” Findings of ACL 2025, 2483–2493. https://aclanthology.org/2025.findings-acl.126/

Huang, Pei, Jiayu Li, Yuhao Zhang, Changliu Liu, and Shih-hsi Alex Liu. 2026. “Parameterized Abstract Interpretation for Transformer Verification.” Proceedings of AAAI 2026. https://doi.org/10.1609/aaai.v40i42.40860

Jia, Robin, Aditi Raghunathan, Kerem Göksel, and Percy Liang. 2019. “Certified Robustness to Adversarial Word Substitutions.” Proceedings of EMNLP-IJCNLP 2019. Association for Computational Linguistics. https://aclanthology.org/D19-1423/

Karthikeyan, T, Om Dehlan, Mausam, and Manish Gupta. 2025. “LRPLAN: A Multi-Agent Collaboration of Large Language and Reasoning Models for Planning with Implicit and Explicit Constraints.” Findings of EMNLP 2025, 8280–8310. https://doi.org/10.18653/v1/2025.findings-emnlp.440

Kartáč, Ivan, Kristýna Onderková, Jan Bronec, Zdeněk Kasner, Mateusz Lango, and Ondřej Dušek. 2026. “UFAL-CUNI at SemEval-2026 Task 11: An Efficient Modular Neuro-Symbolic Method for Syllogistic Reasoning.” Proceedings of the 20th International Workshop on Semantic Evaluation. Association for Computational Linguistics. https://aclanthology.org/2026.semeval-1.418/

King, James C. 1976. “Symbolic Execution and Program Testing.” Communications of the ACM 19(7): 385–394. https://doi.org/10.1145/360248.360252

Kirtania, Shashank, Priyanshu Gupta, and Arjun Radhakrishna. 2024. “LOGIC-LM++: Multi-Step Refinement for Symbolic Formulations.” Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations, 56–63. https://aclanthology.org/2024.nlrse-1.6/

Kuhn, Tobias. 2014. “A Survey and Classification of Controlled Natural Languages.” Computational Linguistics 40(1): 121–170. https://doi.org/10.1162/COLI_a_00168

Lalwani, Vivek, et al. 2025. “Autoformalizing Natural Language to First-Order Logic: A Case Study in Logical Fallacy Detection.” Findings of IJCNLP-AACL 2025, 132–147. https://aclanthology.org/2025.findings-ijcnlp.8/

Lee, Kang-il, Segwang Kim, and Kyomin Jung. 2023. “Weakly Supervised Semantic Parsing with Execution-based Spurious Program Filtering.” Proceedings of EMNLP 2023, 6884–6894. https://aclanthology.org/2023.emnlp-main.425/

Li, Yihe, Ruijie Meng, and Gregory J. Duck. 2025. “Large Language Model Powered Symbolic Execution.” Proceedings of the ACM on Programming Languages 9 (OOPSLA2), Article 385, 3148–3176. https://doi.org/10.1145/3763163

Liang, Chen, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. 2017. “Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision.” Proceedings of ACL 2017. Association for Computational Linguistics. https://aclanthology.org/P17-1003/

Liu, Jiangming, Shay B. Cohen, and Mirella Lapata. 2018. “Discourse Representation Structure Parsing.” Proceedings of ACL 2018. Association for Computational Linguistics. https://aclanthology.org/P18-1040/

Locascio, Nicholas, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2018. “Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge.” Proceedings of EMNLP 2018. Association for Computational Linguistics. https://aclanthology.org/D18-1436/

MacCartney, Bill, and Christopher D. Manning. 2007. “Natural Logic for Textual Inference.” Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing. https://aclanthology.org/W07-1401/

MacCartney, Bill, and Christopher D. Manning. 2009. “An Extended Model of Natural Logic.” Proceedings of IWCS 2009. https://aclanthology.org/W09-3714/

Manhaeve, Robin, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. “DeepProbLog: Neural Probabilistic Logic Programming.” Advances in Neural Information Processing Systems 31. https://proceedings.neurips.cc/paper/2018/hash/dc5d637ed5e62c36ecb73b654b05ba2a-Abstract.html

Mao, Ziyu, Jingyi Wang, Jun Sun, Shengchao Qin, and Jiawen Xiong. 2025. “LLM-Aided Automatic Modeling for Security Protocol Verification.” Proceedings of the 47th IEEE/ACM International Conference on Software Engineering, 2025. https://ink.library.smu.edu.sg/sis_research/10297/

Mitchell, Jacqueline L., Brian Hyeongseok Kim, Chenyu Zhou, and Chao Wang. 2025. “Can LLMs Formally Reason as Abstract Interpreters for Program Analysis?” arXiv:2503.12686. https://arxiv.org/abs/2503.12686

Mou, Lili, Zhengdong Lu, Hang Li, and Zhi Jin. 2017. “Coupling Distributed and Symbolic Execution for Natural Language Queries.” Proceedings of ICML 2017. https://proceedings.mlr.press/v70/mou17a.html

Olausson, Theo X., Alex Gu, Benjamin Lipkin, Cedegao E. 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.” Proceedings of EMNLP 2023. Association for Computational Linguistics. https://aclanthology.org/2023.emnlp-main.313/

Pan, Liangming, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. “Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning.” Findings of EMNLP 2023. Association for Computational Linguistics. https://aclanthology.org/2023.findings-emnlp.248/

Pasupat, Panupong, and Percy Liang. 2016. “Inferring Logical Forms from Denotations.” Proceedings of ACL 2016. Association for Computational Linguistics. https://aclanthology.org/P16-1004/

Pei, Yu, Yongping Du, and Xingnan Jin. 2025. “FoVer: First-Order Logic Verification for Natural Language Reasoning.” Transactions of the Association for Computational Linguistics 13: 1340–1359. https://aclanthology.org/2025.tacl-1.61/

Tafjord, Oyvind, Bhavana Dalvi Mishra, and Peter Clark. 2021. “ProofWriter: Generating Implications, Proofs, and Abductive Statements over Natural Language.” Findings of ACL-IJCNLP 2021. Association for Computational Linguistics. https://aclanthology.org/2021.findings-acl.317/

Tantakoun, Nat, Christian Muise, and Yilun Zhu. 2025. “Large Language Models as Planning Formalizers: A Survey.” Findings of ACL 2025. https://aclanthology.org/2025.findings-acl.1291/

Tziafas, Georgios, and Hamidreza Kasaei. 2026. “Neuro-Symbolic Task Planning and Object Manipulation with Large Language Models.” Autonomous Robots 50. https://doi.org/10.1007/s10514-026-10230-6

Wang, Bailin, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2018. “Execution-Guided Neural Program Decoding.” Proceedings of the First Workshop on Learned Program Synthesis. https://arxiv.org/abs/1807.03100

Wang, Boxin, Chejian Xu, Xiangyu Liu, Yu Cheng, and Bo Li. 2023. “Robustness-Aware Word Embedding Improves Certified Robustness to Adversarial Word Substitutions.” Findings of ACL 2023. Association for Computational Linguistics. https://aclanthology.org/2023.findings-acl.620/

Wang, Jiayimei, Tao Ni, Wei-Bin Lee, and Qingchuan Zhao. 2025. “A Contemporary Survey of Large Language Model Assisted Program Analysis.” Trends in Artificial Intelligence. https://arxiv.org/abs/2502.18474

Wang, Wenguan, Yi Yang, and Fei Wu. 2025. “Towards Data- and Knowledge-Driven AI: A Survey on Neuro-Symbolic Computing.” IEEE Transactions on Pattern Analysis and Machine Intelligence 47(2): 878–899. https://doi.org/10.1109/TPAMI.2024.3483273

Wang, Yu, You Zhang, Hao Zhang, and Dan Xu. 2026. “YNU-NLP at SemEval-2026 Task 11: A Neuro-Symbolic Approach with Reflexion Mechanism Disentangling Content and Formal Reasoning in Language Models.” Proceedings of the 20th International Workshop on Semantic Evaluation. Association for Computational Linguistics. https://doi.org/10.18653/v1/2026.semeval-1.126

Weng, Ke, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. 2025. “Autoformalization in the Era of Large Language Models: A Survey.” arXiv:2505.23486. https://arxiv.org/abs/2505.23486

Xia, Shihao, Mengting He, Shuai Shao, Tingting Yu, Yiying Zhang, Nobuko Yoshida, and Linhai Song. 2026. “SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models.” Proceedings of the ACM on Programming Languages 10 (OOPSLA1), Article 109. https://doi.org/10.1145/3798217

Yang, Zonglin, et al. 2024. “Autoformalization of Natural Language Reasoning Problems to First-Order Logic.” Proceedings of EMNLP 2024. Association for Computational Linguistics. https://aclanthology.org/2024.emnlp-main.1132/

Ye, Mao, Chengyue Gong, and Qiang Liu. 2020. “SAFER: A Structure-Free Approach for Certified Robustness to Adversarial Word Substitutions.” Proceedings of ACL 2020. Association for Computational Linguistics. https://aclanthology.org/2020.acl-main.317/

Ye, Xi, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. “SatLM: Satisfiability-Aided Language Models Using Declarative Prompting.” Advances in Neural Information Processing Systems 36. https://arxiv.org/abs/2305.09656

Zheng, Zihan, Tianle Cui, Chuwen Xie, Jiahui Pan, Qianglong Chen, and Lewei He. 2025. “PlanningArena: A Modular Benchmark for Multidimensional Evaluation of Planning and Tool Learning.” Proceedings of ACL 2025, 31047–31086. https://doi.org/10.18653/v1/2025.acl-long.1499