DOC2SPEC: SYNTHESIZING FORMAL PROGRAMMING SPECIFICATIONS FROM NATURAL LANGUAGE VIA GRAMMAR INDUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Ensuring that API implementations and usage comply with natural language programming rules is critical for software correctness, security, and reliability. Formal verification can provide strong guarantees but requires precise specifications, which are difficult and costly to write manually. To address this challenge, we present Doc2Spec, a multi-agent framework that automatically induces a domain-specific grammar from natural-language API rules and uses it to guide specification generation. Doc2Spec fixes a domain-agnostic logical skeleton as a grammar template, prompts LLMs to infer domain-specific predicates and sorts, and formalizes each rule within the resulting grammar, turning an unreliable one-shot translation into a sequence of constrained, checkable steps. Across six benchmarks spanning Solidity and Rust, Doc2Spec improves precision by 0.28 and recall by 0.37 over baselines that lack grammar induction or perform it in one unstaged step, demonstrating the benefits of grammar-guided formalization and the staged pipeline. Moreover, its formalized rules enable symbolic-execution tools to uncover 142 previously unknown rule violations, confirming the rules' correctness and practical usefulness.

1 INTRODUCTION

Modern software systems are increasingly constructed by composing and reusing modules and libraries (Mikkonen & Taivalsaari, 2019; Saied et al., 2018). These modules are often developed independently and later integrated into larger systems, making well-defined application programming interfaces (APIs) essential for correctness, security, and maintainability (Lamothe et al., 2021; Rauf et al., 2019). Beyond function signatures, APIs also specify semantic requirements for correct implementation and proper use (De, 2017; Cummaudo et al., 2019). These requirements are usually written in natural language in API documentation or code comments and include constraints on inputs and outputs, program state changes, and error handling (Pandita et al., 2012; Zhang et al., 2021).

Violating API requirements can cause serious security and reliability issues. For example, the ERC20 standard in Figure 1 specifies rules for implementing public smart contract APIs (Vogelsteller & Buterin, 2015). A violation, such as failing to check caller privileges in `transferFrom()` as required by the highlighted sentence, can let attackers steal digital assets. Similar problems occur in other systems, such as Rust's `unsafe` APIs, where incorrect use can lead to safety or correctness violations (*e.g.*, out-of-bounds, invalid free) (Qin et al., 2020). Ensuring that API implementations and their usage follow the documented natural-language rules is therefore critical.

Formal verification is a principled way to ensure rule compliance, using techniques such as symbolic execution (Asadian et al., 2022; 2024), model checking (Nehai et al., 2018; Martinelli et al., 2019), and deductive verification (Summers & Müller, 2020; Anureev et al., 2019). These methods provide mathematically grounded guarantees that an implementation satisfies a specification. However, they require precise formal specifications from developers to describe the intended code behaviors. Writing such specifications is difficult, as it demands expertise in formal methods, deep knowledge of the code, and careful treatment of corner cases. As a result, despite their strength, formal verification techniques are still hard to adopt in practice (Khazeev et al., 2020; Reid et al., 2020).

To address this challenge, recent work has explored using large language models (LLMs) to automatically synthesize formal specifications from natural language. At a high level, this task can be viewed

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

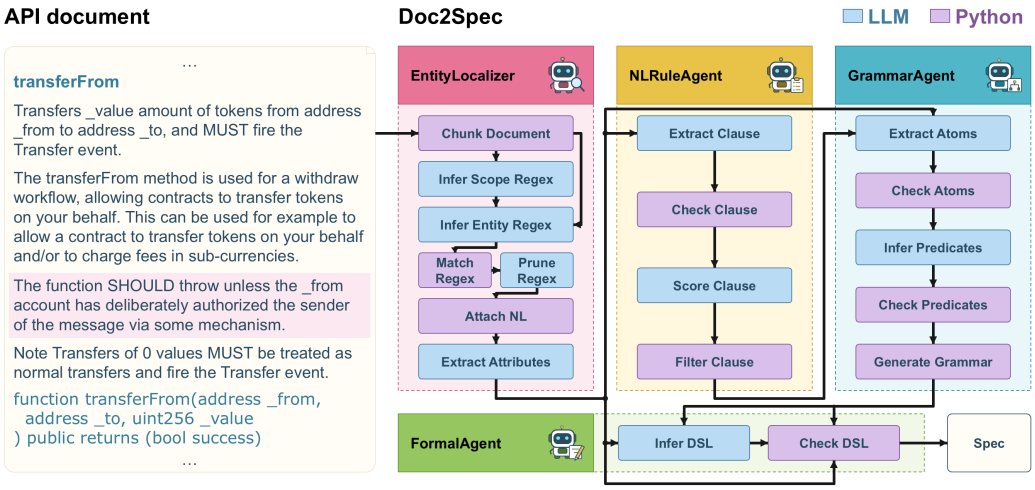


Figure 1: Workflow of Doc2Spec. *(LLM-based components are in blue, and Python tools are in purple. Example outputs of the four agents are shown in Figures 4–7 in the appendix.)*

as a translation problem, and LLMs have shown strong ability to map between different semantic representations (Zhu et al., 2024; Schneider et al., 2024). Existing approaches generally fall into two categories. The first uses in-context learning, where carefully designed prompts (Shi et al., 2023; Wei et al., 2022) or multi-agent pipelines (Zhang et al., 2024; 2025) guide LLMs to generate specifications (Ye et al., 2023; Wu et al., 2022; Misu et al., 2024; Pan et al., 2023; Kirtania et al., 2024; Yao et al., 2023; Kamath et al., 2023; Wu et al., 2024; Sun et al., 2025; Mugnier et al., 2025; Loughridge et al., 2025; Olausson et al., 2023; Wen et al., 2024; Ma et al., 2025a; Ryu et al., 2025; Yang et al., 2025; Sun et al., 2026; 2024). Although effective in some settings, these methods often require substantial domain-specific customization and do not reliably generalize to new or generic scenarios. The second category relies on post-training or fine-tuning LLMs (Hahn et al., 2022; Chen et al., 2023; 2025; Azerbayev et al., 2023; Yang et al., 2023; Zrelli et al., 2024; Polu et al., 2023; First et al., 2023; Chakraborty et al., 2025). However, in many cases, the required labeled data is scarce, costly to obtain, or unavailable.

This raises a fundamental question: how can we build a general technique to translate natural language programming rules into formal specifications across diverse coding scenarios, without relying on post-training or domain-specific customization?

As shown in Figure 1, we address this problem by inducing a grammar of formal specifications from the natural-language rules and using it to guide specification generation. Our key insight is that a formal specification rests on three parts: a general logical skeleton shared across domains (conditions, obligations, and how they combine), a domain-specific vocabulary of predicates and sorts, and the content of each individual specification. Rather than producing all three at once, our approach builds them in order: each part constrains the next, so every LLM step works within a fixed space rather than an unconstrained one. Because the intermediate results are explicit and structured, each step can also be checked deterministically, turning an unreliable one-shot translation into a sequence of verifiable steps.

Building on this insight, we propose Doc2Spec, a multi-agent framework that takes natural-language documents of programming rules as input and generates specifications suitable for formal verification (*e.g.*, symbolic execution). Doc2Spec operates in four stages: identifying constrained entities, extracting rule-related sentences, inducing a grammar from them, and using the grammar to formalize the rules. The key step is grammar induction: we design a domain-agnostic grammar template and prompt an LLM to summarize the domain-specific predicates and sorts from the natural-language rules, which together with the template form a domain-specific grammar. Unlike CLOVER (Ryu et al., 2025), which extracts predicates but relies on a fixed set of sorts, Doc2Spec infers sorts dynamically for each domain, making it more flexible across domains.

We evaluate Doc2Spec on six documents of natural-language rules, covering both API implementation and API usage and spanning two programming languages (Solidity and Rust). We compare Doc2Spec with three baselines: one that formalizes rules in a single prompt, one that induces the grammar in a single prompt and then uses the grammar to formalize rules, and one that relies on a manually crafted grammar for rule formalization (Xia et al., 2026). Doc2Spec improves precision by 0.28 and recall by 0.37 over the first two baselines, highlighting the benefits of grammar-guided formalization and its staged agent pipeline. Compared with the third baseline, Doc2Spec achieves higher recall on two benchmarks despite requiring no manual grammar engineering. Moreover, we integrate the formalized rules with downstream symbolic-execution tools: they uncover 142 previously unknown rule violations in real-world Solidity smart contracts and open-source Rust crates, with only 21 false alarms, demonstrating both the correctness and the practical utility of the generated specifications.

In sum, we make the following contributions:

- We propose a novel approach for automatically inducing grammars from natural language rules.
- We design Doc2Spec, a multi-agent system that induces grammars from API rules and uses them to translate the rules into formal specifications for verification.
- We evaluate Doc2Spec on six benchmarks, demonstrating its effectiveness and the practical utility of its formalized rules.

2 Background and Related Work

This section provides background on this project, including natural-language API usage and implementation rules, as well as prior work on using LLMs to automate formal verification.

2.1 Natural Language API Rules

Modules and libraries expose APIs that allow external code to access their functionality. These APIs are typically accompanied by documentation or comments that describe how to use them correctly, including valid inputs, call order, return-value checks, and calling context. For example, comments before the declaration of Rust's standard library function `dealloc(ptr: *mut u8, layout: Layout)` specify that `ptr` cannot be `NULL`. Otherwise, using the function with a `NULL` pointer introduces a serious security vulnerability.

Some APIs also serve as public standards and are implemented by multiple groups in different versions. In such cases, the API documentation specifies mandatory requirements that implementations must follow to ensure consistent and expected behavior across versions. Figure 1 illustrates this scenario using ERC20, a public standard for Solidity smart contracts that manage digital tokens on Ethereum. Once a contract is deployed, anyone can invoke its public function `transferFrom()`. Therefore, the function must verify the caller's privileges as required by the highlighted sentence. Otherwise, any user could steal tokens by using this function to transfer tokens to his own address.

2.2 Automated Formal Verification via LLMs

Existing techniques that use LLMs to automate or assist with formal software verification mainly focus on generating verification specifications and proofs. Traditionally, these artifacts were written manually, which is tedious and requires substantial expertise in both formal verification and the software's domain. Broadly, existing techniques can be divided into two groups.

First, some researchers treat LLMs as black boxes and apply in-context learning to generate specifications and proofs. Various strategies have been proposed to improve performance, including few-shot prompting (Ye et al., 2023; Wu et al., 2022), chain-of-thought prompting (Misu et al., 2024), feedback from prover error messages (Pan et al., 2023; Kirtania et al., 2024; Yao et al., 2023; Kamath et al., 2023; Wu et al., 2024; Sun et al., 2025; Mugnier et al., 2025; Loughridge et al., 2025), majority voting over multiple specification generations for the same input (Olausson et al., 2023), iterative generation until prover acceptance (Wen et al., 2024), structure-first specification synthesis (Ryu et al., 2025), mutation and ranking of erroneous LLM-generated specifications (Ma et al., 2025a), summarizing common mistakes and refining generation errors via a multi-agent system (Yang et al., 2025; Sun et al., 2026), and joint generation of code and specifications with cross-checking (Sun

et al., 2024). While effective, these methods are often domain-specific and rely heavily on carefully engineered prompts and examples, limiting their generalization to new domains.

Second, other researchers post-train or fine-tune open-source LLMs for formal verification tasks, achieving success on Verus (Chen et al., 2025), Lean (Azerbayev et al., 2023; Yang et al., 2023; Polu et al., 2023), regular expressions (Hahn et al., 2022), temporal logic (Chen et al., 2023), computation tree logic (Zrelli et al., 2024), Isabelle/HOL (First et al., 2023), and F* (Chakraborty et al., 2025). However, these techniques require substantial amounts of paired natural-language descriptions and corresponding specifications or proofs. Such labeled data are scarce in many domains (Chen et al., 2025; 2023), limiting the feasibility of fine-tuning open-source LLMs in practice.

3 Problem Formulation

We study the problem of extracting code entities, their usage or implementation rules, and a formal rule grammar from unstructured natural language documents.

Let $D = \{l_1, l_2, \ldots, l_n\}$ denote a natural language document consisting of text lines. We assume no prior knowledge of the document's domain, structure, or programming content. Given D, Doc2Spec identifies a set of *entities* $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$, where each entity corresponds to an API, struct, or similar code element, and may have multiple attributes. Doc2Spec also induces a rule grammar $\mathcal{G}$ in Extended Backus–Naur form (EBNF) and extracts a set of rules $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}$, where each rule applies to a specific entity and is valid under $\mathcal{G}$. We formalize the problem as learning a mapping

$$f : D \longrightarrow (\mathcal{T}, \mathcal{G}, \mathcal{R}).$$

For instance, when processing the document in Figure 1, Doc2Spec induces a grammar for it. It identifies the function `transferFrom()` in the document as a program entity, extracts the natural-language rules constraining it (including the highlighted sentence), and formalizes these rules according to the induced grammar.

4 Design of Doc2Spec

This section presents the detailed design of Doc2Spec, which takes an API document as input and operates in four stages: (1) locating code entities, (2) identifying natural language statements that specify usage or implementation rules, (3) inducing a grammar in EBNF from the rules, and (4) formalizing the rules using the grammar.

Figure 1 shows the overall architecture of Doc2Spec. Doc2Spec is domain-agnostic: neither its pipeline nor its prompt design assumes any domain knowledge. Even the grammar begins as a basic template that captures only general logical structure, which is then filled with domain-specific predicates and sorts inferred automatically from the natural-language rules in the input document. Moreover, Doc2Spec decomposes the task top-down, from document to rules, and then from each rule to atoms and predicates. Because each step performs a single, simple operation, a deterministic Python tool can verify its output before the next step begins (*e.g.*, checking that the extracted text is faithful to the document or that all inputs have been processed), thereby improving overall correctness. The appendix contains all prompt templates for Figure 1 and the example LLM responses.

4.1 Stage 1: Localizing Programming Entities

EntityLocalizer identifies the programming entities in each input document, attaches its natural-language description to each entity, and extracts its declaration attributes. As shown in Figure 1, it combines four LLM passes with three lightweight Python tools for data preprocessing and output validation. EntityLocalizer's design addresses three key challenges.

First, an input document may be too long for an LLM to process in one prompt, and even when it fits, directly enumerating entities can be inconsistent: the same LLM may identify an entity at one occurrence but miss it at another. To address this, we split each document into 80-line chunks (***Chunk Document***). For each chunk, we ask the LLM to infer regular expressions that match the first line of an entity declaration (***Infer Entity Regex***). A Python tool then applies these regular expressions to the chunk, ensuring that matching entities are extracted consistently (***Match Regex***).

Second, programs are hierarchical, and functions in different scopes (*e.g.*, interfaces, structs, and classes) may have the same name. Without identifying an entity's scope, downstream verification tools cannot determine which entity a generated formal rule belongs to. To address this, we use a separate LLM pass to infer scope patterns (***Infer Scope Regex***): a regular expression matching the first line of a scope and capturing its name, together with the delimiters that open and close its body. We apply these patterns to the document and track the delimiters to determine the line range of each scope. This allows us to place each matched entity in its scope chain. We infer scope patterns before entity patterns. We provide the scope patterns, along with examples, to the entity pass. This helps the LLM distinguish scope syntax from entity syntax.

Third, inferred entity patterns may be incorrect or may match lines that are not entities governed by natural-language rules (*e.g.*, import statements, code quoted in comments). We therefore validate the patterns in two steps. (1) A Python tool (***Match Regex***) checks that each pattern compiles and matches its example, and then finds all matching lines. (2) The LLM reviews each pattern and all of its matches and decides whether to keep it (***Prune Regex***). We keep a pattern if at least one match is a genuine entity declaration and discard it only if all matches are irrelevant.

After validating the entity patterns, a Python tool (***Attach NL***) attaches the natural-language block immediately preceding each identified entity, following the common convention that an entity's description appears before its declaration. Finally, an LLM pass (***Extract Attributes***) extracts the entity's attributes from its complete declaration and description, including its kind, name, ordered parameters and their types, return values, modifiers, visibility, enclosing scopes, and attached description, for subsequent agents. Figure 4 (appendix) shows the function `transferFrom()` in Figure 1 extracted as an entity, together with its description and attributes.

4.2 Stage 2: Extracting Natural-Language Rules

NLRuleAgent takes each entity produced by EntityLocalizer, together with its declaration and attached natural-language description, and extracts from the description the sentences that state implementation or usage rules. As shown in Figure 1, *NLRuleAgent* contains two LLM passes with two Python tools.

The main challenge is that extracting rules directly from free-form text is unreliable in two ways. (1) The LLM may paraphrase, merge, or invent text instead of reporting what the document says. (2) The LLM may be inconsistent about which clauses count as rules[1]. We address this by separating *finding* candidate clauses from *deciding* whether each is a rule, with a deterministic check in between.

First, the LLM extracts every clause in the input description that may state an independent rule for the entity (***Extract Clause***). As shown in Listing 14 in the appendix, the prompt requires each clause to express exactly one independently testable requirement: a sentence with multiple requirements is split into separate clauses, while a condition is kept with its consequence. The prompt also clarifies what does not count as a clause, excluding topic summaries, definitions, rationale, examples, and parameter descriptions. To reduce missed rules, the LLM must inspect every sentence, list item, and structured field in the input. Finally, it must copy each clause verbatim, which keeps the extracted clauses faithful to the document.

Second, a Python tool validates the returned clauses against the input description (***Check Clause***). It first normalizes both sides by lower-casing text, collapsing whitespace, standardizing quotation marks and dashes, and removing Markdown backticks. It then checks that each clause appears as a substring of the description. Clauses that the LLM altered or fabricated are discarded. The tool also removes duplicates and clauses that are fully contained in another retained clause.

Third, the LLM scores each remaining clause on whether it states a rule for the target entity (***Score Clause***). Each clause is evaluated independently twice ($i \in \{1, 2\}$), producing a label r_i ($r_i = 1$ for a rule, $r_i = -1$ otherwise) and a confidence score $c_i \in [0, 1]$. As shown in Listing 18 in the appendix, the prompt defines what counts as a rule and which cases to exclude, following the same criteria as ***Extract Clause***. The prompt also emphasizes that the score indicates the LLM's confidence that the clause is a rule, rather than the strength or importance of the rule itself.

Finally, a Python tool computes a score using the following formula and treats a clause with a score greater than 0.5 as a rule (***Filter Clause***). For example, Figure 5 (appendix) shows the two confidence

[1]We call a text span that potentially states a rule a *clause*, and a clause confirmed to state a rule a *rule*.

```
<spec>    ::= <rule>+
<rule>    ::= [<modality>] <formula> [if <formula>] ;
<modality>::= MODALITY
<formula> ::= <predicate> | not <formula> | (<formula>) | <formula> <op> <formula>
<op>      ::= and | or
<predicate> ::= PREDICATE([<sorts>])
<sorts>   ::= <sort> | <sort>, <sorts>
<sort>    ::= SORT | string | number | boolean | TargetAttr(string) | TargetReturn([number])
```

Figure 2: Simplified DSL Template. *(Nonterminals are enclosed in angle brackets, all-uppercase tokens are placeholders instantiated during grammar induction, and all other tokens are terminals written literally. TargetAttr(string) selects an attribute of the target, and TargetReturn([number]) selects a return value of the target.)*

scores reported by the LLM for the highlighted sentence in Figure 1, together with the score computed by the tool. Since the computed score exceeds 0.5, the sentence is identified as a rule.

$$\text{score}(s) \;=\; \frac{1}{1+\exp\Big(-\sum_{i=1}^{2} r_i c_i\Big)}.$$

4.3 Stage 3: Inducing Rule Grammar

GrammarAgent induces an EBNF grammar by filling in a fixed DSL template rather than generating one from scratch. We observe that most API rules state that a property holds under a condition, where both the property and the condition are expressed with domain-specific predicates. The condition is optional, since some rules apply unconditionally, and rules carry different levels of obligation (*e.g.*, "must", "may"), so we include a modality indicating whether a violation is an error or a warning. Figure 2 shows the resulting template. It fixes this common structure but leaves the predicates and their sorts unspecified, since they vary across domains. Grammar induction thus reduces to constructing the domain-specific vocabulary of predicates and sorts.

GrammarAgent processes the rules one at a time, each with the associated entity's declaration. Generating a predicate directly from a rule is unreliable, because a rule often combines multiple requirements and a condition. Naming a predicate before this structure is disentangled may make it too large and specific to that one rule, or even reverse the rule's meaning (*e.g.*, turning "throw unless authorized" into "throw if authorized"). *GrammarAgent* therefore uses two LLM passes. It first splits each rule into atoms, which are independently checkable conditions or obligations, and the Boolean formulas connecting them (*Extract Atoms*). It then names each atom as a predicate and gives every argument a sort (*Infer Predicates*). A Python tool checks each pass (*Check Atoms*, *Check Predicates*), verifying that formulas refer to the correct atoms and that all atoms are translated, and a final tool collects the predicates and sorts to build the grammar (*Generate Grammar*).

The prompt for *Extract Atoms* is shown in Listing 23 (appendix). It defines an atom as one independently falsifiable observable relation and gives detailed guidance on what should become an atom and how a rule should be split. For example, when one relation can hold while another fails, it directs the LLM to split the two into separate atoms. As another example, it requires each atom to be grounded in a concrete observation (*e.g.*, a state transition or an emitted effect). The prompt also guards against common mistakes. Finally, it requires the LLM to label each atom as an obligation or a condition and to output the Boolean formulas combining them into the rule's logical structure, which the last agent later uses to formalize the rule.

The prompt for *Infer Predicates* is shown in Listing 28 (appendix). It instructs the LLM to translate the extracted atoms into the smallest vocabulary of predicates and sorts. It requires a one-to-one mapping in which every atom yields exactly one predicate. Each predicate inherits its atom's classification, becoming an obligation or condition predicate that guides formal rule generation. Each predicate and sort also carries a natural-language description. The prompt further enforces consistency: a parameter name that is shared across atoms and denotes one semantic variable must take the same sort wherever it occurs, and every referenced sort must be declared in the same response. Finally, it forbids encoding concrete values or the whole rule into a predicate name, keeping predicate names small and general.

Table 1: Precision and recall of rule formalization. *(The Doc2Spec columns use GPT-OSS-20B; SiO and GiO use GPT-5.6. SymGPT has no Rust results.)*

API Doc	Comparison with Baselines								Comparison with LLM Backbones					
	GiO		SiO		SymGPT		Doc2Spec		OSS-120B		DeepSeek-V4		GPT-5.6	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
ERC20	0.38	0.56	0.56	0.79	1.00	0.72	0.79	0.90	0.72	0.83	0.79	0.83	0.91	1.00
ERC721	0.42	0.80	0.53	0.94	0.94	0.77	0.56	0.78	0.51	0.74	0.68	0.67	0.59	0.77
ERC1155	0.11	0.63	0.13	0.67	1.00	0.80	0.37	0.73	0.41	0.83	0.49	0.80	0.47	0.80
Rust-1	0.81	0.38	0.75	0.23			0.80	0.78	0.74	0.77	0.71	0.72	0.82	0.91
Rust-2	0.75	0.28	0.73	0.09			0.82	0.77	0.81	0.77	0.78	0.75	0.85	0.89
Rust-3	0.53	0.75	0.46	0.67			0.61	0.75	0.60	0.70	0.60	0.73	0.76	0.92
Average	**0.45**	**0.41**	**0.43**	**0.30**			**0.73**	**0.78**	**0.70**	**0.77**	**0.71**	**0.74**	**0.78**	**0.89**

For example, as shown in Figure 6 (appendix), the highlighted rule in Figure 1 yields two predicates and one sort. The obligation predicate `ThrowObligation()` denotes that the function reverts execution, the condition predicate `AuthorizedCondition(Address, Address)` denotes that the first address has authorized the second, and the single sort `Address` represents an Ethereum address.

4.4 Stage 4: Formalizing Rules

FormalAgent formalizes each natural-language rule into one DSL statement, using the predicates, sorts, and Boolean formulas extracted from the rule, together with the rule's associated entity and its attributes. As specified by the prompt in Listing 34 (appendix), the statement must follow the structure in Figure 2: the LLM does not re-derive the logical structure but fills in the extracted predicates and prefixes the appropriate modality. The LLM further instantiates each predicate by replacing the sorts in its signature with concrete arguments. A Python tool built on the Lark parser then checks that the DSL statement conforms to the grammar (*Check DSL*). For example, Figure 7 (appendix) shows the DSL statement that *FormalAgent* generates for the highlighted sentence in Figure 1, where the sorts of `AuthorizedCondition()` are instantiated with the concrete parameters `_from` and `sender` of `transferFrom()`.

5 Evaluation

Benchmarks. We evaluate Doc2Spec using six API documents: three ERC standards and three Rust source files. The ERC standards specify how to implement public APIs for smart contracts managing fungible tokens, non-fungible tokens (NFTs), and both. They are also studied in SymGPT (Xia et al., 2026), whose authors identify 90 implementation rules. The three Rust files are from the Rust standard library and cover pointer arithmetic and memory allocation. Their API comments contain 425 natural-language rules on how these APIs should be used. Together, the benchmarks span two programming languages (Solidity and Rust) and cover both API implementation rules and API usage rules. Table 2 (appendix) shows more detailed information.

Baselines. We compare Doc2Spec against three baselines. (1) *Spec-in-One* (SiO) asks a commercial LLM (GPT-5.6) to identify and formalize all rules in a single prompt, with no grammar or staged decomposition. (2) *Grammar-in-One* (GiO) first summarizes a grammar in one prompt, then uses it to formalize the rules with FormalAgent. (3) *SymGPT* (Xia et al., 2026) formalizes ERC rules using manually crafted grammars. SiO measures the benefit of grammar guidance, GiO isolates the benefit of staged grammar induction, and SymGPT provides a comparison with hand-written grammars. The prompts for SiO and GiO are in Section H.6 (appendix).

We do not compare with NL2Type (Malik et al., 2019), TypeWriter (Pradel et al., 2020), REQ2LTL (Ma et al., 2025b), or CLOVER (Ryu et al., 2025) because these tools either focus on inferring parameter and return types, focus on formalizing system-level rules rather than API rules, or support only a limited set of predefined sorts.

LLMs. Unless otherwise stated, Doc2Spec uses GPT-OSS-20B, a 20B-parameter open-source model (OpenAI, 2025). To study how the backbone affects Doc2Spec, we also evaluate it with

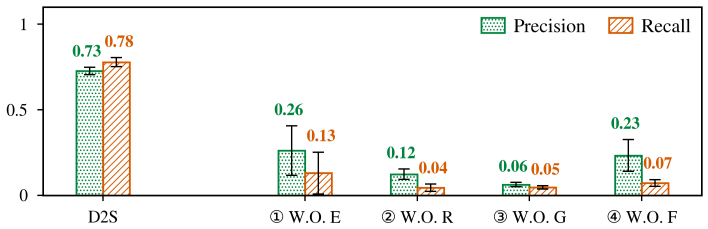


Figure 3: Doc2Spec ablation results. *(W.O.: without; E: EntityLocalizer; R: NLRuleAgent; G: GrammarAgent; F: FormalAgent. Each ablation replaces one agent with a single prompt.)*

GPT-OSS-120B (OpenAI, 2025), DeepSeek-V4 (DeepSeek-AI, 2026), and the frontier model GPT-5.6 (OpenAI, 2026). These cover scalability within a model family, generalizability across families, and frontier-level models.

Metrics. We compare the formalized rules against a gold-standard set of natural-language rules. For the three ERCs, we reuse the rule set from SymGPT. For the Rust documents, we manually construct the gold-standard set from each document. Since the tool produces formal rules, at least two authors manually judge whether each one correctly formalizes a gold-standard rule, following a shared rubric; the agreement rate between authors is 87% (Cohen's $\kappa = 0.75$). A gold-standard rule is a true positive (TP) if fully matched and a false negative (FN) otherwise, and an unmatched formal rule is a false positive (FP). We then compute precision and recall accordingly. The full rubric is in Section C (appendix).

Configurations. When Doc2Spec detects errors in an LLM output with its Python tools, it prompts the LLM to retry, up to three times. The temperature is set to 1 and the reasoning effort is set to medium for all experiments. Each configuration is run *three* times, and we report the average.

5.1 Effectiveness

As shown in Table 1, Doc2Spec achieves an average precision of 0.73, ranging from 0.37 on ERC1155 to 0.82 on Rust-2, and an average recall of 0.78, with every benchmark reaching at least 0.73. We observe no clear precision–recall trade-off and no correlation between document length and performance. The three ERCs consist of API implementation rules and the three Rust documents of API usage rules. Doc2Spec handles both: recall is comparable across the two, and is slightly higher on the implementation rules.

Compared with GiO and SiO, Doc2Spec is substantially better on both metrics: it reaches 0.73 precision and 0.78 recall, versus 0.45/0.41 for GiO and 0.43/0.30 for SiO. This holds even though both baselines use a stronger backbone (GPT-5.6) than Doc2Spec (GPT-OSS-20B). Against SiO, Doc2Spec improves precision on all six benchmarks and recall on all but ERC721; against GiO, it improves precision on five and recall on four. These gains show the benefit of *grammar-guided formalization* and of *inducing the grammar through staged decomposition* rather than in one step.

Compared with SymGPT, Doc2Spec achieves lower precision across all three ERCs, as SymGPT is primarily a bug-detection tool focused on result accuracy. However, Doc2Spec attains higher recall on ERC20 and ERC721, for two reasons. First, SymGPT does not define a grammar for return-value rules, whereas Doc2Spec infers one and can thus formalize the rules with it. Second, SymGPT avoids designing grammars for underspecified rules (*e.g.*, "throw on any error"), whereas Doc2Spec can infer grammars for such rules. Overall, these recall gains over SymGPT show that *Doc2Spec's automatic grammar induction can match or exceed hand-written grammars in many cases.*

Finally, we evaluate Doc2Spec with different backbones. Precision ranges from 0.70 to 0.78 and recall from 0.74 to 0.89. Consistency across model sizes and families indicates that *Doc2Spec's stability comes from its staged, grammar-guided design rather than any single backbone.*

5.2 Ablation Analysis

To understand each agent's contribution, we ablate one agent at a time by replacing it with a single-prompt alternative and measuring how Doc2Spec's precision and recall change over the six

benchmarks. For EntityLocalizer, we give the LLM the full document and ask it to identify all entities directly. For NLRuleAgent, we provide the document and extracted entities and ask it to identify the rule-stating sentences for each entity. For GrammarAgent, we provide all rules and the DSL template and ask it to summarize the sorts and predicates. For FormalAgent, we provide all entities, attributes, rules, and the full grammar in one prompt and ask it to produce the formalized DSL rules.

As shown in Figure 3, removing any single agent sharply degrades Doc2Spec. Precision drops to 0.26, 0.12, 0.06, and 0.23 without EntityLocalizer, NLRuleAgent, GrammarAgent, and FormalAgent respectively, and recall to 0.13, 0.04, 0.05, and 0.07. These results show that all four agents contribute substantially and *validate the effectiveness of Doc2Spec's multi-agent design*.

5.3 Downstream Verification

To show that the rules formalized by Doc2Spec are correct and useful, we use existing general-purpose symbolic execution engines (SymGPT's backend for Solidity smart contracts and KLEE (Cadar et al., 2008) for Rust) to check real code against the rules and detect real violations.

Our evaluated code set contains 200 Ethereum smart contracts (181 for ERC20, 12 for ERC721, and 7 for ERC1155) and 20 Rust crates from crates.io that either implement the APIs or exercise the APIs our specifications cover. The formalized rules cannot be checked directly: a symbolic execution engine needs to know when a violation occurs, and some predicates refer to program states that do not have corresponding program variables. We therefore use an LLM to instrument each program with the required ghost variables and assertions before running the engine. The full pipeline is described in Appendix D.

We compare against nine baselines. (1) *LLM-on-Docs* asks GPT-5.6 to check each pair of a natural-language rule document and a source file and judge whether any rule is violated. (2) *LLM-on-Rules* instead gives GPT-5.6 each rule formalized by Doc2Spec, paired with each source file, and asks the same question, isolating the effect of formalization. (3)–(9) SymGPT and six static Rust bug-detection tools (Semgrep, LockBud, Prusti, Rudra, MIRAI, and MirChecker). Detailed baseline information is in Table 3 (appendix). We count real code sites where a rule is violated (TP) and false alarms (FP). As shown in Table 5 (appendix), Doc2Spec with symbolic execution detects 142 real violations with only 21 false alarms, confirming *the formalized rules' correctness and practical usefulness.*

Compared with our approach, the baselines LoD and LoR report many true positives, and LoR even detects more than Doc2Spec. However, they produce hundreds of false alarms (324 and 602, respectively), which demonstrates that our pipeline is essential for precise detection of rule violations. Doc2Spec trails SymGPT (268 TPs, 6 FPs) on the ERC benchmarks, but SymGPT relies on a hand-written grammar, whereas Doc2Spec induces the grammar fully automatically. On Rust, Doc2Spec detects 24 violations with no false alarms, exceeding all six specialized tools combined (13 TPs, 90 FPs); these tools must encode each checked rule through dedicated program analysis and thus scale poorly to new rules.

6 Conclusion

This paper presents Doc2Spec, a multi-agent framework that automatically formalizes natural-language rules on API implementation and usage. Our key insight is to decompose specification generation into ordered, constrained steps: Doc2Spec fixes a domain-agnostic logical skeleton as a grammar template, induces domain-specific predicates and sorts from the natural-language rules to form a grammar, and uses the grammar to guide rule formalization.

Across six benchmarks spanning Solidity and Rust and covering both API implementation and API usage, Doc2Spec improves precision by 0.28 and recall by 0.37 over baselines that lack grammar induction or perform it in one unstaged step. Integrated with downstream symbolic-execution tools, its formalized rules uncover 142 previously unknown rule violations in real-world smart contracts and Rust crates, with only 21 false alarms. These results demonstrate the benefits of grammar induction for LLM-based specification generation. We hope Doc2Spec serves as a step toward making formal verification accessible directly from natural-language documentation.

AI USE STATEMENT

Large language models are components of the proposed Doc2Spec framework and the experimental evaluation. They are used to analyze API documentation, extract natural-language rules, construct domain-specific grammars, and translate rules into formal specifications. The models, prompts, and validation procedures are described in the main text and appendices.

During manuscript preparation, we also used generative AI tools to assist with language editing, literature search, LaTeX formatting, and the preparation and refinement of figures and tables.

The authors reviewed and revised the AI-assisted material, checked technical claims and references against the original sources, and take full responsibility for the final content of this work, including its text, methods, results, and artifacts.

REFERENCES

Igor Anureev, Natalia Garanina, Tatiana Liakh, Andrei Rozov, Vladimir Zyubin, and Sergei Gorlatch. Two-step deductive verification of control software using Reflex. In *Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI '19)*, Novosibirsk, Russia, 2019.

Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. Applying symbolic execution to test implementations of a network protocol against its specification. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST '22)*, Virtual Event, 2022.

Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. Monitor-based testing of network protocol implementations using symbolic execution. In *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES '24)*, Vienna, Austria, 2024.

Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti project: Formal verification for Rust. In *Proceedings of the 14th International Symposium on NASA Formal Methods (NFM '22)*, Pasadena, CA, USA, 2022.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, Virtual Event, Germany, 2021.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, USA, 2008.

Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*, Ottawa, Ontario, Canada, 2025.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong Zhou. Automated proof generation for Rust code via self-evolution. In *Proceedings of the 13th International Conference on Learning Representations (ICLR '25)*, Singapore, 2025.

Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. NL2TL: Transforming natural languages to temporal logics using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*, Singapore, 2023.

Alex Cummaudo, Rajesh Vasa, and John Grundy. What should I document? A preliminary systematic mapping study into API documentation knowledge. In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*, Porto de Galinhas, Pernambuco, Brazil, 2019.

Brajesh De. API documentation. In *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, pp. 59–80. Apress, 2017.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. ERC-721: Non-fungible token standard, 2018. https://eips.ethereum.org/EIPS/eip-721.

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*, San Francisco, USA, 2023.

Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. Formal specifications from natural language, 2022. URL https://arxiv.org/abs/2206.01962.

Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

Mansur Khazeev, Manuel Mazzara, Hamna Aslam, and Daniel de Carvalho. Towards a broader acceptance of formal verification tools. In *Proceedings of the 22nd International Conference on Interactive Collaborative Learning (ICL '2019)*, Bangkok, Thailand, 2020.

Shashank Kirtania, Priyanshu Gupta, and Arjun Radhakrishna. LOGIC-LM++: Multi-step refinement for symbolic formulations. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (NLRSE '24)*, Bangkok, Thailand, 2024.

Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of API evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, Virtual Event, Republic of Korea, 2021.

Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=yBgTVWccIx.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE '25)*, Ottawa, Ontario, Canada, 2025a.

Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification: Automated translation of software requirements to LTL via hierarchical semantics decomposition using LLMs. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE '25)*, Seoul, South Korea, 2025b.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE '19)*, Montreal, Canada, 2019.

Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Albina Orlando, Antonella Santone, and Gigliola Vaglini. Model checking based approach for compliance checking. *Information Technology and Control*, 48(2):278–298, 2019.

Tommi Mikkonen and Antero Taivalsaari. Software reuse in the era of opportunistic design. *IEEE Software*, 36(3):105–111, 2019.

MIRAI contributors. MIRAI: Rust mid-level IR abstract interpreter, 2026. URL https://github.com/endorlabs/MIRAI. Accessed September 18, 2026.

Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-assisted synthesis of verified Dafny methods. In *Proceedings of the ACM on Software Engineering (FSE '24)*, New York, USA, 2024.

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Yuanyuan Zhou. Laurel: Unblocking automated verification with large language models. In *Proceedings of the ACM on Programming Languages (OOPSLA '25)*, New York, USA, 2025.

Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. Model-checking of smart contracts. In *Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings), Green Computing and Communications (GreenCom), Cyber, Physical and Social Computing (CPSCom), and Smart Data (SmartData) (iThings/GreenCom/CPSCom/SmartData '18)*, Halifax, NS, Canada, 2018.

Theo X. Olausson, Alex Gu, Ben Lipkin, Cedegao E. Zhang, Armando Solar-Lezama, Joshua B. Tenenbaum, and Roger P. Levy. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*, Singapore, 2023.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

OpenAI. GPT-5.6 Sol. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026. Accessed: 2026-09-21.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*, Singapore, 2023.

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland, 2012.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR '23)*, Kigali, Rwanda, 2023.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*, Seoul, South Korea, 2020.

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, 2020.

Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. Understanding and detecting real-world safety issues in Rust. *IEEE Transactions on Software Engineering*, 50(6):1306–1324, 2024. doi: 10.1109/TSE.2024.3380393. URL https://doi.org/10.1109/TSE.2024.3380393.

Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. ERC-1155: Multi token standard, 2018. https://eips.ethereum.org/EIPS/eip-1155.

Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of API usability evaluation methods. *Computer Science Review*, 33:49–68, 2019.

Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are. *arXiv preprint arXiv:2010.16345*, 2020.

Hyun Ryu, Gyeongman Kim, Hyemin S. Lee, and Eunho Yang. Divide and translate: Compositional First-Order logic translation and verification for complex logical reasoning. In *Proceedings of the 13th International Conference on Learning Representations (ICLR '25)*, Singapore, 2025.

Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164–179, 2018.

Phillip Schneider, Manuel Klettner, Kristiina Jokinen, Elena Simperl, and Florian Matthes. Evaluating large language models in semantic parsing for conversational question answering over knowledge graphs. *arXiv preprint arXiv:2401.01711*, 2024.

Semgrep contributors. Semgrep: Lightweight static analysis for many languages, 2026. URL `https://github.com/semgrep/semgrep`. Accessed September 18, 2026.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning (ICML '23)*, Honolulu, HI, 2023.

Alexander J Summers and Peter Müller. Automating deductive verification for weak-memory programs (extended version). *International Journal on Software Tools for Technology Transfer*, 22 (6):709–728, 2020.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *Proceedings of the International Symposium on AI Verification (SAIV '24)*, Montreal, Canada, 2024.

Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark Barrett, David Dill, Xiaokang Qiu, and Shuvendu K Lahiri. ClassInvGen: Class invariant synthesis using large language models. In *Proceedings of the International Symposium on AI Verification (SAIV '25)*, Zagreb, Croatia, 2025.

Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill, and Clark Barrett. VeriStruct: AI-assisted automated verification of data-structure modules in Verus. In *Proceedings of the 32nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '26)*, Turin, Italy, 2026.

The Rust Project. Rust core library: `alloc/global.rs`, 2025a. URL `https://github.com/rust-lang/rust/blob/1.87.0/library/core/src/alloc/global.rs`.

The Rust Project. Rust core library: `ptr/const_ptr.rs`, 2025b. URL `https://github.com/rust-lang/rust/blob/1.87.0/library/core/src/ptr/const_ptr.rs`.

The Rust Project. Rust core library: `ptr/mut_ptr.rs`, 2025c. URL `https://github.com/rust-lang/rust/blob/1.87.0/library/core/src/ptr/mut_ptr.rs`.

Fabian Vogelsteller and Vitalik Buterin. ERC-20: Token standard, 2015. https://eips.ethereum.org/EIPS/eip-20.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*, New Orleans, LA, USA, 2022.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV '24)*, Montreal, QC, Canada, 2024.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *Proceedings of the International Conference on Learning Representations (ICLR '24)*, Vienna, Austria, 2024.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS '22)*, New Orleans, USA, 2022.

Shihao Xia, Mengting He, Shuai Shao, Tingting Yu, Yiying Zhang, Nobuko Yoshida, and Linhai Song. SymGPT: Auditing smart contracts via combining symbolic execution with large language models. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1):482–512, 2026. doi: 10.1145/3798217. URL https://doi.org/10.1145/3798217.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated proof generation for Rust code. *Proc. ACM Program. Lang.*, (OOPSLA2), 2025.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in Rust. *arXiv preprint arXiv:2311.03739*, 2023.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23)*, Taipei, Taiwan, 2023.

Jing-Xuan Zhang, Chuan-Qi Tao, Zhi-Qiu Huang, and Xin Chen. Discovering API directives from API specifications with text classification. *Journal of Computer Science and Technology*, 36(4): 922–943, 2021.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. In *International Conference on Machine Learning (ICML '24)*, 2024.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *International Conference on Machine Learning (ICML '25)*, 2025.

Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. Multilingual machine translation with large language models: Empirical results and analysis. In *Findings of the Association for Computational Linguistics: 2024 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL '24)*, Mexico City, Mexico, 2024.

Rim Zrelli, Henrique Amaral Misson, Maroua Ben Attia, Felipe Gohring de Magalhaes, Abdo Shabah, and Gabriela Nicolescu. Advancing formal verification: Fine-tuning LLMs for translating natural language requirements to CTL specifications. In *Proceedings of the 2024 International Workshop on Rapid System Prototyping (RSP '24)*, Raleigh, NC, USA, 2024.

Appendix

A Agent Outputs

EntityLocalizer Output Example

Target declaration

```
function transferFrom(address _from, address _to, uint256 _value)
  public returns (bool success)
```

Attributes `kind: function; name: transferFrom; visibility: public`
Parameters (ordered) `_from: address, _to: address, _value: uint256`
Return `success: bool` **Hierarchy** `[]`
Attached NL
Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the Transfer event.

The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD throw unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

Figure 4: EntityLocalizer output for the highlighted example in Figure 1.

NLRuleAgent Output Example

The function SHOULD throw unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

Candidate `candidate-0003`

Score Clause Evaluation 1: $r_1 = +1$ (`true`), $c_1 = 0.8$; Evaluation 2: $r_2 = +1$ (`true`), $c_2 = 0.8$.

Filter Clause $\text{score}(s) \approx 0.8320 > 0.5$.

Result Retain the clause as a rule.

Figure 5: NLRuleAgent output for the highlighted example in Figure 1.

GrammarAgent Output Example

Sort `Address`: Ethereum address

Semantic atoms → predicate signatures

```
O_throw      -> ThrowObligation()
C_authorized -> AuthorizedCondition(_from: Address, sender: Address)
```

Outcome formula `O_throw` **Condition formula** `not C_authorized`

Figure 6: GrammarAgent output for the highlighted example in Figure 1.

FormalAgent Output Example

```
SHOULD ThrowObligation() if not AuthorizedCondition(_from, sender);
```

Figure 7: FormalAgent output for the highlighted example in Figure 1.

B BENCHMARKS

Table 2: Benchmark Information.

API Doc	Lines	Entities	Rules
ERC20 (Vogelsteller & Buterin, 2015)	88	11	21
ERC721 (Entriken et al., 2018)	436	19	41
ERC1155 (Radomski et al., 2018)	710	12	28
Rust-1 (The Rust Project, 2025b)	1,741	47	168
Rust-2 (The Rust Project, 2025c)	2,157	62	226
Rust-3 (The Rust Project, 2025a)	277	5	31
Total	**5,409**	**156**	**515**

C RULE EVALUATION RUBRIC

Each generated DSL rule is paired with an extracted natural-language (NL) clause and a target entity. Reviewers use the NL clause to identify candidate gold-standard requirements, then check entity grounding and the actual DSL semantics. The procedure below uses the gold-to-generated direction when naming match cardinalities.

1. **Find candidate matches using the extracted NL clauses.** Compare each extracted clause with the gold-standard NL rules. Identify candidates that describe the same requirement, or complementary parts of that requirement. Compare meaning rather than wording: retain the required behavior, conditions, exceptions, and modality. An extracted clause with no corresponding gold requirement cannot yield an accepted match.
2. **Check the associated entity.** For each candidate, verify the declaration, declaration attributes, and lexical hierarchy against the gold target. Names alone are insufficient: distinguish overloads, enclosing scopes, parameter and return roles, and receiver information where relevant. Reject candidates bound to the wrong entity. Entity correctness is a prerequisite, not a separate counting unit.
3. **Determine the candidate match.** After entity matching, distinguish one gold rule paired with one generated rule (one-to-one), one gold rule split across several generated rules (one-to-many), and several gold rules merged into one generated rule (many-to-one). Use only rules needed to express the candidate requirement; do not combine unrelated targets or add duplicates to a group.
4. **Validate the DSL for the identified case.** Missing, malformed, or placeholder DSL cannot participate in an accepted match. Interpret parseable DSL using its predicate definitions. In every case, check predicate meaning; argument roles, bindings, sorts, constants, and negation; the presence and composition of conditions; Boolean operators and scope; modality; and all consequences, exceptions, and required ordering. Reject omissions, contradictions, unsupported additions, strengthening, or weakening. Apply these checks as follows:

   One-to-one.
   The generated DSL must express the complete meaning of the gold rule. Matching the attached NL clause or a predicate name is insufficient.

   One-to-many.
   The smallest necessary set of generated DSL rules must jointly express the complete gold requirement. Every required consequence must retain its own condition and modality; splitting a rule must not make a conditional obligation unconditional. Partial coverage is not a TP.

   Many-to-one.
   The single generated DSL rule must fully express each gold requirement in the group, preserving each condition–consequence association and modality. Do not credit several gold rules merely because their topics appear in one extracted NL clause. If only a subset is covered, accept that subset only when the complete DSL is equivalent to it and contains no unsupported obligations.

   Repartition a more complex candidate into these cases only when the resulting groups independently satisfy all checks; do not accept an otherwise unresolved many-to-many group.

5. **Assign TP, FP, and FN.** Assign each gold rule and each generated rule to at most one accepted group. Count each covered gold rule as one TP and each uncovered gold rule as one FN. Count each generated rule outside accepted groups as one FP, including unnecessary duplicates, invalid DSL, incorrect targets, and predictions without a gold requirement. Thus, a valid one-to-many group contributes one TP; a valid many-to-one group covering k gold rules contributes k TPs. A rejected candidate leaves its gold rules as FNs unless another valid group covers them, and its generated rules as FPs unless accepted elsewhere.

D INTEGRATION WITH SYMBOLIC EXECUTION ENGINES

The pipeline takes the rules formalized by Doc2Spec and the program code as input. It contains two agents that process each pair of an input program and a formalized rule in turn. The first agent inserts the ghost variables that represent the program states required by the rule, together with the updates to these variables along the program. The second agent instruments assertions at the appropriate program sites, so that an assertion fails exactly when the rule is violated. The symbolic execution engine, either SymGPT's backend or KLEE, then searches for an input that triggers each inserted assertion. If such an input is found, a rule violation is detected; otherwise, the program is verified to satisfy the rule.

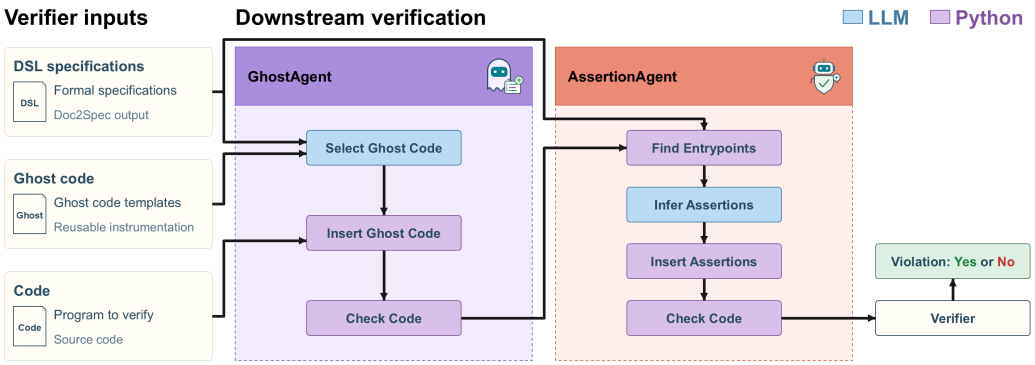


Figure 8: Integration with downstream symbolic execution engines.

E BASELINES FOR RULE VIOLATION DETECTION

Table 3: Baselines used in the downstream rule-violation detection. *(Capability summarizes their primary analysis focus. Language indicates the programming language analyzed by each tool. Stars reports the number of GitHub stars. A gray cell denotes an unavailable GitHub star count.)*

Tool	Capability	Language	Stars
LLM-on-Docs	Rule compliance	Solidity & Rust	
LLM-on-Rules	Rule compliance	Solidity & Rust	
SymGPT (Xia et al., 2026)	ERC compliance	Solidity	8
Semgrep (Semgrep contributors, 2026)	API misuse	Rust	16,725
LockBud (Qin et al., 2024)	Concurrency	Rust	591
Prusti (Astrauskas et al., 2022)	Correctness	Rust	1,811
Rudra (Bae et al., 2021)	Memory safety	Rust	1,367
MIRAI (MIRAI contributors, 2026)	Correctness	Rust	229
MirChecker (Li et al., 2021)	Memory safety	Rust	165

F RUST CRATE BENCHMARKS

Table 4: Information about the Rust crates. *(Stars reports the number of GitHub stars. A gray cell denotes an unavailable GitHub star count.)*

Crate (version)	Stars
aegis (0.9.12)	54
cyberex (0.3.12)	
falcon_capstone (0.5.2)	4
fast-able (1.11.17)	
fast-float (0.2.0)	292
grubbnet (0.1.7)	77
halo2curves-axiom (0.7.0)	7
hugepage-rs (0.1.0)	13
i_tree (0.9.0)	1
i_triangle (0.26.0)	160
jh (0.1.0)	2,266
jomini (0.27.0)	90
lurk-cli (0.3.8)	1,154
meshopt (0.3.0)	223
ptr_hash (0.1.1)	103
qwt (0.2.0)	57
revm-interpreter (10.0.3)	2,243
roqoqo-quest (0.14.4)	9
splay_tree (0.3.1)	19
stackvector (1.1.1)	3

G DOWNSTREAM RULE-VIOLATION DETECTION

Table 5: Violation detection results. *(LoD denotes LLM-on-Docs, LoR denotes LLM-on-Rules, Semg denotes Semgrep, LockB denotes LockBud, MirC denotes MirChecker, and D2S denotes Doc2Spec. x_y denotes x true violations and y false alarms. A gray cell denotes an unsupported programming language.)*

	LoD	LoR	SymGPT	Semg	LockB	Prusti	Rudra	MIRAI	MirC	D2S
ERC	60_{79}	222_{483}	268_{6}							118_{21}
Rust	24_{245}	18_{119}		0_{12}	0_{0}	0_{0}	6_{18}	1_{54}	6_{6}	24_{0}
Total	$\mathbf{84_{324}}$	$\mathbf{240_{602}}$	$\mathbf{268_{6}}$	$\mathbf{0_{12}}$	$\mathbf{0_{0}}$	$\mathbf{0_{0}}$	$\mathbf{6_{18}}$	$\mathbf{1_{54}}$	$\mathbf{6_{6}}$	$\mathbf{142_{21}}$

H PROMPT TEMPLATES AND EXAMPLES

H.1 ENTITYLOCALIZER

H.1.1 INFER SCOPE REGEX

Prompt template | Infer Scope Regex

```
You can perform the following actions.
Available Actions:
- record_lexical_patterns: Record new lexical-scope patterns.
  Input Schema: {
  "type": "object",
  "properties": {
    "hierarchy_patterns": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "kind": {
            "type": "string"
          },
          "pattern": {
            "type": "string",
            "description": "Anchored scope-start regex containing a Python-style named capture group (?P<name>...)."
          },
          "example": {
            "type": "string"
          },
          "open_pattern": {
            "type": "string",
            "description": "Regex for a lexical nesting opener, for example \\{."
          },
          "close_pattern": {
            "type": "string",
            "description": "Regex for the matching lexical nesting closer, for example \\}."
          }
        },
        "required": [
          "kind",
          "pattern",
          "example",
          "open_pattern",
          "close_pattern"
        ]
      }
    }
  },
  "required": [
    "hierarchy_patterns"
  ]
}
- no_new_lexical_patterns: Confirm that the chunk has no lexical-scope syntax.
  Input Schema: {
  "type": "object",
  "properties": {
    "reason": {
      "type": "string"
    }
  },
  "required": [
    "reason"
  ]
}


Analyze this chunk of a programming API or technical specification during the scope pass.

hierarchy_patterns match the first line of a lexical scope that may contain API declarations. A semantic container is a named
or receiver-bound declaration group whose identity is needed to distinguish declarations nested inside it, such as a namespace
, type, interface, service, extension, or receiver/type-bound implementation group. The executable body of one callable, a
control-flow block, and an anonymous block are NOT semantic containers. Its regex MUST contain exactly one Python-style named
capture group `(?P<name>...)`, whose value is the COMPLETE source-level qualifier used to distinguish a nested member. For a
receiver/type-bound group, capture the complete receiver/type expression while excluding only the group's introducer and
generic binder; do not capture merely its final identifier or type argument. Also return regexes that count the scope's
lexical nesting opener and matching closer. For brace-delimited scopes these are `\{` and `\}`. Do not put a specific
observed scope name in a regex.

All start regexes are candidate generators, so favor recall, but every regex must:
- be anchored with ^ and work against a line after leading/trailing whitespace is stripped;
- describe declaration syntax rather than a specific scope name;
- not match comment markers or ordinary prose;
- include one exact example line from this chunk, without a line-number prefix.

Before returning, mechanically check that every regex matches its own example and that each relevant semantic-container
declaration in this chunk is matched by a scope regex; do not infer coverage merely because two declarations look similar.

Chunk:
```
<<DOCUMENT_CHUNK>>
```
```

Prompt template | Infer Scope Regex *(continued)*

```
Discover only hierarchy_patterns.
Return exactly one action: call record_lexical_patterns with hierarchy_patterns, or call no_new_lexical_patterns when this
chunk contains no lexical-scope syntax.
Never return a blank response or an empty actions list.
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 1: The prompt template of *Infer Scope Regex*. *(For parameters involved in the template,* ***DOCUMENT_CHUNK*** *represents the document chunk to analyze.)*

Input | Document chunk

```
...
Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.

The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be
used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function
SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
...
```

Listing 2: Example for parameter **DOCUMENT_CHUNK** in Listing 1.

LLM response | Infer Scope Regex

```
{
  "actions": [
      {
          "name": "no_new_lexical_patterns",
          "input": {
              "reason": "No scope declarations are present in the provided chunk."
          }
      }
  ]
}
```

Listing 3: Example LLM response for *Infer Scope Regex* using the prompt template in Listing 1.

The ERC-20 example contains no enclosing lexical scope. A supplementary ERC-721 example with a non-empty scope result appears in Section H.5.

H.1.2 Infer Entity Regex

Prompt template | Infer Entity Regex

```
You can perform the following actions.
Available Actions:
- record_lexical_patterns: Record new target-entity patterns.
  Input Schema: {
  "type": "object",
  "properties": {
    "entity_patterns": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "kind": {
            "type": "string"
          },
          "pattern": {
            "type": "string"
          },
          "example": {
            "type": "string"
          }
        },
        "required": [
```

Prompt template | Infer Entity Regex *(continued)*

```
          "kind",
          "pattern",
          "example"
        ]
      }
    }
  },
  "required": [
    "entity_patterns"
  ]
}
- no_new_lexical_patterns: Confirm that the chunk has no entity-declaration syntax.
  Input Schema: {
  "type": "object",
  "properties": {
    "reason": {
      "type": "string"
    }
  },
  "required": [
    "reason"
  ]
}

Analyze this chunk of a programming API or technical specification during the entity pass.

An entity is a declaration to which behavioral documentation can attach, such as a callable,
operation, constructor, event, type, or another declared API element. Do not impose a fixed list
of entity kinds: infer what the document treats as entities.

entity_patterns match the FIRST PHYSICAL LINE of target entity declarations.
Existing scope patterns provide context only; they do not cover entity declarations.

All start regexes are candidate generators, so favor recall, but every regex must:
- be anchored with ^ and work against a line after leading/trailing whitespace is stripped;
- describe declaration syntax rather than a specific entity name;
- not match comment markers or ordinary prose;
- include one exact example line from this chunk, without a line-number prefix.

For entity_patterns, prefer the shortest declaration prefix that distinguishes the syntax. Stop at a stable token such as the
declaration name or opening parameter delimiter. Do not encode a particular return type, terminator, body opener, or the
absence/presence of a body when the same entity kind can vary there. Before returning, mechanically check that every regex
matches its own example and that each declaration in this chunk is actually matched by a new entity regex; do not infer
coverage merely because two declarations look similar.

Existing scope patterns:
```json
<<EXISTING_PATTERNS_JSON>>
```

Chunk:
```
<<DOCUMENT_CHUNK>>
```

Discover only entity_patterns.
Hierarchy-pattern coverage NEVER counts as entity-pattern coverage. A semantic container can also
be a documented target entity (for example an API type, interface, module, or service), so in the entity pass independently
emit an entity regex for such declarations when the document treats the container itself as an entity, even if the same first
line already matches a hierarchy pattern.
Return exactly one action: call record_lexical_patterns with entity_patterns, or call no_new_lexical_patterns when this chunk
contains no entity-declaration syntax.
Never return a blank response or an empty actions list.
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 4: The prompt template of *Infer Entity Regex*. *(For parameters involved in the template,* ***EXISTING_PATTERNS_JSON*** *represents the scope patterns discovered by* *Infer Scope Regex*, *and* ***DOCUMENT_CHUNK*** *represents the document chunk to analyze.)*

Input | Existing patterns

```
{
  "hierarchy_patterns": []
}
```

Listing 5: Example for parameter **EXISTING_PATTERNS_JSON** in Listing 4.

Input | Document chunk

```
Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.

The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be
used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function
SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 6: Example for parameter **DOCUMENT_CHUNK** in Listing 4.

LLM response | Infer Entity Regex

```
{
  "actions": [
    {
      "name": "record_lexical_patterns",
      "input": {
        "entity_patterns": [
          {
            "kind": "function",
            "pattern": "^function\\s+[A-Za-z_][A-Za-z0-9_]*\\b",
            "example": "function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)"
          }
        ]
      }
    }
  ]
}
```

Listing 7: Example LLM response for *Infer Entity Regex* using the prompt template in Listing 4.

H.1.3 PRUNE REGEX

Prompt template | Prune Regex

```
You can perform the following actions.
Available Actions:
- record_pattern_validation: Keep or reject every candidate-generator pattern after reviewing its full-document matches.
  Input Schema: {
  "type": "object",
  "properties": {
    "decisions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "keep": {
            "type": "boolean"
          },
          "reason": {
            "type": "string"
          }
        },
        "required": [
          "id",
          "keep",
          "reason"
        ]
      }
    }
  },
  "required": [
    "decisions"
  ]
}


Validate these entity candidate-generator patterns against every place each regex matches in the
full document. A valid entity pattern has at least one match that is a declaration to which this document attaches behavioral
documentation. Candidate generators may be broad because the later classifier rejects individual false matches.

Keep a broad pattern when at least one of its matches is a genuine target and later per-candidate classification can safely
reject the others. Reject a pattern only when its matches are exclusively the wrong syntax, such as imports, local statements,
 prose, or non-semantic implementation scopes.
Use the complete match list rather than trusting the pattern's positive example. Return every supplied pattern ID exactly once
 in one `record_pattern_validation` action.

Full-document match evidence:
```json
<<FULL_DOCUMENT_MATCH_EVIDENCE_JSON>>
```
```

Prompt template | Prune Regex *(continued)*

```
```
Analyze the request step by step, then respond with JSON in the following format
{{
    "analysis": "Your analysis here",
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 8: The prompt template of *Prune Regex*. *(For parameters involved in the template,* ***FULL_DOCUMENT_MATCH_EVIDENCE_JSON*** *represents candidate patterns and all their matches in the document.)*

**Input** | Match evidence


```
[
  {
    "id": "entity-pattern-0001",
    "pattern": {
      "kind": "function",
      "pattern": "^function\\s+[A-Za-z_][A-Za-z0-9_]*\\b",
      "example": "function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)"
    },
    "full_document_match_count": 1,
    "all_matches": [
      {
        "line": 7,
        "text": "function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)"
      }
    ]
  }
]
```


Listing 9: Example for parameter **FULL_DOCUMENT_MATCH_EVIDENCE_JSON** in Listing 8.

LLM response | Prune Regex


```
{
  "analysis": "The provided pattern 'entity-pattern-0001' matches a Solidity function declaration: `function transferFrom(
address _from, address _to, uint256 _value) public returns (bool success)`. This is a true structural declaration that could
attach behavioral documentation. Since there is at least one match that is a meaningful entity definition, the pattern should
be kept for further classification.",
  "actions": [
    {
      "name": "record_pattern_validation",
      "input": {
        "decisions": [
          {
            "id": "entity-pattern-0001",
            "keep": true,
            "reason": "Matches a Solidity function declaration, which is a valid entity definition for attaching behavioral
documentation."
          }
        ]
      }
    }
  ]
}
```


Listing 10: Example LLM response for *Prune Regex* using the prompt template in Listing 8.

H.1.4 Extract Attributes

Prompt template | Extract Attributes

```
You can perform the following actions.
Available Actions:
- record_candidate_analysis: Classify every candidate ID and extract attributes for accepted entities.
  Input Schema: {
  "type": "object",
  "properties": {
    "entities": {
      "type": "array",
      "items": {
```
```

Prompt template | Extract Attributes *(continued)*

```
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "is_entity": {
            "type": "boolean"
          },
          "declaration_end_line": {
            "type": "integer",
            "description": "Exact one-based last physical line of the declaration header."
          },
          "hierarchy": {
            "type": "array",
            "description": "Corrected containing semantic-scope names, outermost first.",
            "items": {
              "type": "string"
            }
          },
          "attributes": {
            "type": "object",
            "description": "Extracted entity attributes; empty for rejected candidates."
          }
        },
        "required": [
          "id",
          "is_entity",
          "declaration_end_line",
          "hierarchy",
          "attributes"
        ]
      }
    }
  },
  "required": [
    "entities"
  ]
}


Classify every candidate below according to what this document treats as target entities. Scope patterns were learned and
validated before entity patterns. The computed hierarchy is therefore a strong suggestion, but you must repair it using `
enclosing_scope_evidence` when a captured name is incomplete or wrong. For a receiver/type-bound declaration group, use its
complete receiver or type expression, excluding the group introducer and any generic binder. Preserve outermost-to-innermost
order and do not add a scope unsupported by the evidence.
Hierarchy contains only semantic containers whose names distinguish the nested declaration; never include an ordinary function
/callable body. Reject imports, function calls, local statements, prose, and code merely quoted inside a comment.

A declaration that introduces a lexical scope is not accepted or rejected merely for being a scope. Accept it only when it is
independently one of this document's target entities; member documentation belongs to the member candidate, not automatically
to the containing scope.

For every candidate, return `declaration_end_line`. For an accepted entity this is the exact last physical line of the
declaration header: include multiline generic parameters, parameters, results, where/constraint clauses, and a terminal body
opener or semicolon, but exclude statements inside the body. For a rejected candidate, return its supplied `declaration_line`
as the end line.
The end line must lie between `declaration_line` and `context_end_line`, inclusive.

For every accepted entity, return the corrected hierarchy and extract useful attributes supported by the complete entity
declaration, such as kind, name, parameters, return type, modifiers, visibility, mutability, or event fields. Use an empty
attributes object and empty hierarchy for a rejected candidate. Do not invent IDs, line numbers, or container names. Return
every supplied ID exactly once.

Candidates:
```json
<<CANDIDATE_RECORDS_JSON>>
```
Analyze the request step by step, then respond with JSON in the following format
{{
    "analysis": "Your analysis here",
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 11: The prompt template of *Extract Attributes*. *(For parameters involved in the template, **CANDIDATE_RECORDS_JSON** represents candidate declarations with line numbers, attached documentation, and enclosing-scope evidence.)*

Input | Candidate records


```
[
  {
    "id": "candidate-0001",
    "candidate_start_line": 7,
    "declaration_line": 7,
    "provisional_entity": "function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)",
    "numbered_declaration_context": "000007: function transferFrom(address _from, address _to, uint256 _value) public returns
(bool success)",
    "context_end_line": 7,
    "comment_range": [
      1,
      5
    ],
    "comment": "Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.\n\nThe
transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used
for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD
 throw unless the _from account has deliberately authorized the sender of the message via some mechanism.\n\nNote Transfers of
 0 values MUST be treated as normal transfers and fire the Transfer event.",
    "computed_semantic_hierarchy": [],
    "enclosing_scope_evidence": []
  }
]
```


Listing 12: Example for parameter **CANDIDATE_RECORDS_JSON** in Listing 11.

LLM response | Extract Attributes


```
{
  "analysis": "Candidate-0001 is a Solidity function declaration. Since the target entities for the document likely include
functions, and the declaration is syntactically complete, it should be classified as an entity. The end line is the same as
the declaration line (7). The hierarchy is unknown because no enclosing contract/interface evidence is provided, so an empty
array is used. Attributes for the function are extracted directly from the declaration.",
  "actions": [
    {
      "name": "record_candidate_analysis",
      "input": {
        "entities": [
          {
            "id": "candidate-0001",
            "is_entity": true,
            "declaration_end_line": 7,
            "hierarchy": [],
            "attributes": {
              "kind": "function",
              "name": "transferFrom",
              "parameters": [
                {
                  "name": "_from",
                  "type": "address"
                },
                {
                  "name": "_to",
                  "type": "address"
                },
                {
                  "name": "_value",
                  "type": "uint256"
                }
              ],
              "returns": [
                {
                  "name": "success",
                  "type": "bool"
                }
              ],
              "visibility": "public"
            }
          }
        ]
      }
    }
  ]
}
```


Listing 13: Example LLM response for *Extract Attributes* using the prompt template in Listing 11.

H.2 NLRuleAgent

H.2.1 Extract Clause

Prompt template | Extract Clause

```
You can perform the following actions.
Available Actions:
- extract_rule_clauses: Return all verbatim contract clauses for the target, or an empty list.
  Input Schema: {
  "type": "object",
  "properties": {
    "clauses": {
      "type": "array",
      "items": {
        "type": "string",
        "description": "One verbatim contract clause from the attached documentation."
      }
    }
  },
  "required": [
    "clauses"
  ]
}


Extract every independently testable contract clause in the attached documentation that applies to the target.

A contract clause is source text that can be used to decide whether an implementation, execution, or observed value conforms
to the documentation. It may specify what the target does or produces, an observable effect, a required relationship, an
allowed or forbidden case, a condition and its consequence, a failure mode, or a property that must remain true.

Rules:
- Use only the attached text. Do not add conclusions from domain knowledge.
- Copy each clause verbatim; do not paraphrase.
- Perform a coverage pass before answering: check every sentence, list item, return/result field, safety section, panic/error
section, and explicit MUST/SHALL/SHOULD/MAY statement.
- Preserve normative force exactly. Keep MUST, SHOULD, MAY, permissions, and prohibitions in the returned span; never silently
 strengthen or weaken them.
- Preserve the full logical scope of conditions, exceptions, conjunctions, and alternatives. In particular, do not return an
effect without its governing if/when/unless condition.
- Preserve words such as "for example" and "e.g." whenever they distinguish an example from a required constant; an example
value is not itself a fixed requirement.
- A clause may be a sentence, a documentation fragment, a tagged field, or a list item. Modal words are not required.
- Include a concise statement only when it specifies an observable result, state transition, side effect, failure behavior,
invariant, permission, prohibition, or caller/implementation obligation with enough information to judge conformance.
- When the target declaration returns or produces a value, a concise attached description of that value may define the target'
s result even without a verb. Keep it when the returned value relationship is concrete enough to test; do not extend this to a
 topic or capability summary that does not describe an output.
- Require an independently enforceable proposition. A topic, capability summary, label, definition, identifier, value
description, or fragment that merely says what the target is about is not a contract clause.
- Preserve enough source context for the clause to be independently understandable. Keep a condition with its consequence and
keep a governing statement with a dependent item.
- Split source text only when each resulting span still has its own subject or unambiguous context and remains independently
testable.
- Do not merge independent requirements merely because they occur in one paragraph. Return each atomic requirement separately
when the exact source supplies enough context to do so.
- Include every independently testable item in any section whose purpose is to define constraints, required conditions,
guarantees, or failure behavior.
- Exclude headings, metadata, reference information, explanatory notes, rationale, examples, and implementation commentary
that do not themselves impose observable behavior. Exclude ordinary argument and return-value descriptions unless they state a
 concrete result relationship, condition, or guarantee.
- Return an empty list only when no source passage can be used to judge conformance of the target.

Target:
```
<<TARGET_DECLARATION>>
```

Attached documentation:
```
<<ATTACHED_DOCUMENTATION>>
```
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 14: The prompt template of *Extract Clause*. *(For parameters involved in the template,* ***TARGET_DECLARATION*** *represents the associated entity declaration, and* ***ATTACHED_DOCUMENTATION*** *represents the natural-language description attached to the entity.)*

Input | Target declaration

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 15: Example for parameter **TARGET_DECLARATION** in Listing 14.

Input | Attached documentation

```
Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.

The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be
used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function
SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.
```

Listing 16: Example for parameter **ATTACHED_DOCUMENTATION** in Listing 14.

LLM response | Extract Clause

```
{
  "actions": [
    {
      "name": "extract_rule_clauses",
      "input": {
        "clauses": [
          "Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.",
          "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.",
          "The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some
mechanism."
        ]
      }
    }
  ]
}
```

Listing 17: Example LLM response for *Extract Clause* using the prompt template in Listing 14.

H.2.2 SCORE CLAUSE

Prompt template | Score Clause

```
You can perform the following actions.
Available Actions:
- score_rule_candidates: Independently score every extracted candidate as a target contract rule.
  Input Schema: {
  "type": "object",
  "properties": {
    "decisions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "is_rule": {
            "type": "boolean"
          },
          "rule_likelihood": {
            "type": "number"
          },
          "reason": {
            "type": "string"
          }
        },
        "required": [
          "id",
          "is_rule",
          "rule_likelihood",
          "reason"
        ]
      }
    }
  },
  "required": [
    "decisions"
  ]
}
```

Prompt template | Score Clause *(continued)*

```
Independently score every extracted candidate below for whether it is a contract rule for the target.

A candidate qualifies only when it is an independently enforceable proposition: after reading the target and attached
documentation, one can state a concrete observable implementation, execution, state, effect, failure, or result that would
comply with or violate the candidate. Mere relevance to the target is insufficient.

Reject topic and capability summaries, labels, definitions, identifiers or metadata stated for reference, rationale, examples,
 implementation commentary, client guidance, and pure parameter or value descriptions. Precision or inspectability is not
enough: do not invent an implied behavioral requirement merely because a declaration, constant, identifier, or documented fact
 could be checked.
Such metadata qualifies only when the candidate explicitly connects it to required observable target behavior. A parameter
field remains a parameter description even when it describes a valid range; keep it only when it explicitly states the target'
s required response, effect, or failure behavior.

Structured result fields inherit a result relationship from their source context: text copied from a return/result field means
 that the target returns or produces that text's value. Keep it when this inherited relationship is concrete enough to test,
even if the extracted candidate is a short noun phrase. This exception does not promote ordinary labels, parameters, metadata,
 or topic summaries.
Also reject a fragment that names an operation or quantity without specifying the observable relationship that makes it
testable. Modal wording alone does not make explanatory or client-facing text a target contract. Keep concise descriptive
wording when it nevertheless defines a concrete return relationship, state transition, effect, condition, guarantee,
permission, or prohibition of the target.

Score `rule_likelihood` from 0 to 1. Use at least 0.70 only when the candidate itself contains enough contract meaning to
formalize a meaningful conformance test; use less than 0.70 when it is vague, referential, explanatory, or dependent on
unstated meaning. Return every supplied ID exactly once.
The score is confidence that the text is a contract rule, not the strength of that rule. A concrete recommendation, permission
, or prohibition can be a high-confidence rule even when its normative force is weaker than an absolute requirement; do not
lower the score merely because compliance is recommended or optional rather than mandatory.

Target:
```
<<TARGET_DECLARATION>>
```

Attached documentation:
```
<<ATTACHED_DOCUMENTATION>>
```

Candidates:
```json
<<RULE_CANDIDATES_JSON>>
```
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 18: The prompt template of *Score Clause*. *(For parameters involved in the template, **TARGET_DECLARATION** represents the associated entity declaration, **ATTACHED_DOCUMENTATION** represents the natural-language description attached to the entity, and **RULE_CANDIDATES_JSON** represents candidate clauses and their identifiers.)*

Input | Target declaration

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 19: Example for parameter **TARGET_DECLARATION** in Listing 18.

Input | Attached documentation

```
The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
```

Listing 20: Example for parameter **ATTACHED_DOCUMENTATION** in Listing 18.

Input | Candidate clauses

```
[
  {
    "id": "candidate-0003",
    "clause": "The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via
some mechanism.",
    "source_field": null
  }
]
```

Listing 21: Example for parameter **RULE_CANDIDATES_JSON** in Listing 18.

LLM response | Score Clause


```
{
  "actions": [
    {
      "name": "score_rule_candidates",
      "input": {
        "decisions": [
          {
            "id": "candidate-0003",
            "is_rule": true,
            "rule_likelihood": 0.8,
            "reason": "Specifies a behavioral expectation (the function should throw when unauthorized), which can be tested;
although not a MUST, it is still a non\u2011vague, enforceable rule."
          }
        ]
      }
    }
  ]
}
```


Listing 22: Example LLM response for *Score Clause* using the prompt template in Listing 18.

H.3 GRAMMARAGENT

H.3.1 EXTRACT ATOMS

Prompt template | Extract Atoms

```
You can perform the following actions.
Available Actions:
- analyze_semantic_atoms: Atomic truth conditions and their boolean structure
  Input Schema: {
  "type": "object",
  "properties": {
    "atoms": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "role": {
            "type": "string",
            "enum": [
              "obligation",
              "condition"
            ]
          },
          "source_span": {
            "type": "string"
          },
          "description": {
            "type": "string"
          },
          "parameters": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "required": [
          "id",
          "role",
          "source_span",
          "description",
          "parameters"
        ]
      }
    },
    "outcome_formula": {
      "type": "string"
    },
    "condition_formula": {
      "type": "string"
    }
  },
  "required": [
    "atoms",
    "outcome_formula",
    "condition_formula"
  ]
}
```

Prompt template | Extract Atoms *(continued)*

```
Decompose one documented contract rule into its semantic atoms before naming predicates.

Use only the supplied rule and associated entity. Do not add domain knowledge.
The associated entity is an out-of-band scope binding: use it to resolve words such as "this" and parameter references, but do
 not create an atom merely to repeat the entity kind, name, signature, or membership. An operation or effect explicitly
related by the rule, other than the bound entity itself, remains part of the semantics and must not be removed.

An atom is one independently falsifiable observable relation. Split outcomes when one can hold while another fails. Preserve
relational atoms such as exact argument/state correspondence or ordering when that relation is itself the property. Preserve
polarity, pre/post state, success/failure, effects and their arguments, ordering, boundary cases, and all explicit boolean
relationships.
Describe the concrete observation that makes each atom true: for example a return relation, state transition, normal or
abnormal completion, call, or emitted effect. Avoid vague descriptions such as an action being triggered, something being
handled normally, or a requirement being satisfied when the rule states a more precise observable behavior.

Keep a trigger and its required result semantically independent. In particular, when a rule requires an effect because an
operation or state transition occurs, the condition atom must describe the underlying operation/transition without assuming
that effect, while the outcome atom must describe the required effect and its arguments. Never make the condition and outcome
circular paraphrases of the same effect occurrence. Conversely, do not weaken an effect requirement into an unspecified "
trigger" action. Analyses that describe the same named effect as occurring in both a condition and an outcome are rejected.

Result qualifiers introduced by phrases such as "such that" or "so that" are part of the obligation they qualify, never an
antecedent. Distribute a shared result qualifier into each alternative outcome.
For explicit alternatives `A or B`, create one complete atom for A and one for B, distribute any shared qualifier into both
atoms, and join them with `or`.
Never encode alternatives as Boolean parameters of one combined atom. More generally, a parameter denotes an observed
participant or value in the rule, never whether an atom/predicate/alternative is true. A named constant or operation may stay
in the atom description instead of becoming a parameter.

Classify atoms as:
- obligation: a required, forbidden, or permitted observable outcome;
- condition: an antecedent, trigger, precondition, exception, or scope stated by the rule.

Preserve the logical polarity of conditional connectives. In particular, `O unless C` means that O is required when C does not
 hold, so represent it as outcome O with condition formula `not C`; never silently rewrite it as `O if C`.

For every atom, copy the smallest useful contiguous `source_span` verbatim from the rule, give a self-contained operational
description, and list only the semantic participant/value names needed to relate it to other atoms. Parameters are rule-local
semantic variables: reuse exactly the same name in every atom that refers to the same participant/value, and use different
names for distinct participants/values. Atoms are expected and allowed to share variables. Do not use opaque participant names
 such as call, contract, function, event, or entity solely to restate the bound entity.

Examples, illustrations, notes, and explanatory calculations do not create additional normative atoms unless the rule
independently states them as a required or forbidden program observation.

Reference every atom in either `outcome_formula` or `condition_formula`. An atom may appear more than once when the rule's
Boolean relation requires it, for example in an equivalence or truth table. A formula contains only atom IDs,
`and`, `or`, `not`, and parentheses, with the rule's exact Boolean structure.
At least one outcome is required. Return an empty condition formula when there is no antecedent.

Rule ID: <<RULE_ID>>
Rule:
```
<<RULE>>
```
Associated entity (scope only):
```
<<TARGET_DECLARATION>>
```
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 23: The prompt template of *Extract Atoms*. *(For parameters involved in the template, **RULE_ID** represents the rule identifier, **RULE** represents the original natural-language rule, and **TARGET_DECLARATION** represents the associated entity declaration.)*

Input | Rule ID

```
R0012
```

Listing 24: Example for parameter **RULE_ID** in Listing 23.

Input | Rule

```
The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
```

Listing 25: Example for parameter **RULE** in Listing 23.

Input | Target declaration

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 26: Example for parameter **TARGET_DECLARATION** in Listing 23.

LLM response | Extract Atoms

```
{
  "actions": [
    {
      "name": "analyze_semantic_atoms",
      "input": {
        "atoms": [
          {
            "id": "C_authorized",
            "role": "condition",
            "source_span": "the _from account has deliberately authorized the sender of the message via some mechanism",
            "description": "The _from account has deliberately authorized the message sender via some mechanism",
            "parameters": [
              "_from",
              "sender"
            ]
          },
          {
            "id": "O_throw",
            "role": "obligation",
            "source_span": "throw",
            "description": "The function reverts execution",
            "parameters": []
          }
        ],
        "outcome_formula": "O_throw",
        "condition_formula": "not C_authorized"
      }
    }
  ]
}
```

Listing 27: Example LLM response for *Extract Atoms* using the prompt template in Listing 23.

H.3.2 Infer Predicates

Listing 28 shows the prompt template for translating semantic atoms into predicates and sorts.

Prompt template | Infer Predicates

```
You can perform the following actions.
Available Actions:
- define_rule_vocabulary: Define sorts, predicates, and the binding for one rule.
  Input Schema: {
  "type": "object",
  "properties": {
    "sorts": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "description": {
            "type": "string"
          }
        },
        "required": [
          "name",
          "description"
        ]
      }
    },
    "predicates": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "atom_id": {
            "type": "string"
          },
          "name": {
            "type": "string"
          },
          "description": {
```

Prompt template | Infer Predicates *(continued)*

```
            "type": "string"
          },
          "parameters": {
            "type": "object",
            "description": "Ordered mapping of parameter name to sort name."
          }
        },
        "required": [
          "atom_id",
          "name",
          "description",
          "parameters"
        ]
      }
    }
  },
  "required": [
    "sorts",
    "predicates"
  ]
}
```

```
Translate an audited semantic-atom analysis into the smallest logical vocabulary.

Use only the supplied rule and target. Do not add facts from domain knowledge.

Definitions:
- Each obligation atom becomes exactly one primary predicate.
- Each condition atom becomes exactly one condition predicate.
- A sort describes the domain of a predicate parameter.

Requirements:
- Return exactly one predicate for every supplied atom and no other predicates.
- Copy each atom's `id` into the predicate's `atom_id`; roles and boolean structure are already fixed and must not be changed.

- Copy the semantic atom's description verbatim as the predicate description. It is an audited truth condition and must not be
 summarized or rewritten.
- Describe the observable relation itself, not meta-language such as "the obligation is satisfied if" and not truth flags
standing for other predicates or alternatives.
- Do not combine independently falsifiable atoms into one predicate.
- Do not encode the associated target/entity identity as a predicate or parameter. It is supplied to the verifier separately
as scope metadata.
- Preserve every listed semantic parameter with the same name. Do not replace concrete observable roles with opaque call/
contract/function/event objects.
- A repeated parameter name denotes one shared semantic variable. Give it the same sort in every predicate where it occurs.
Never duplicate or rename it per atom.
- Keep predicate parameters minimal: include only participants or values needed to evaluate that relation.
- Use stable, descriptive identifiers matching [A-Za-z_][A-Za-z0-9_]*.
- Prefer UpperCamelCase predicate and sort names and lowerCamelCase parameter names.
- Every predicate parameter must reference a sort returned in the same response.
- Do not encode concrete values or the entire rule into a predicate name.

Rule ID: <<RULE_ID>>
Rule:
```
<<RULE>>
```
Associated target:
```
<<TARGET_DECLARATION>>
```

Audited semantic atoms and formula:
```json
<<SEMANTIC_ANALYSIS_JSON>>
```
Respond with JSON in the following format
{{
    "actions": [
        {{
            "name": "action_name",
            "input": {{
                // action input according to the schema
            }}
        }}
    ]
}}
```

Listing 28: The prompt template of *Infer Predicates*. *(For parameters involved in the template, **RULE_ID** represents the rule identifier, **RULE** represents the original natural-language rule, **TARGET_DECLARATION** represents the associated entity declaration, and **SEMANTIC_ANALYSIS_JSON** represents the validated semantic atoms and Boolean formulas.)*

Input | Rule ID

```
R0012
```

Listing 29: Example for parameter **RULE_ID** in Listing 28.

Input | Rule

```
The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
```

Listing 30: Example for parameter **RULE** in Listing 28.

Input | Target declaration

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 31: Example for parameter **TARGET_DECLARATION** in Listing 28.

Input | Semantic analysis

```
{
  "atoms": [
    {
      "id": "C_authorized",
      "role": "condition",
      "source_span": "the _from account has deliberately authorized the sender of the message via some mechanism",
      "description": "The _from account has deliberately authorized the message sender via some mechanism",
      "parameters": [
        "_from",
        "sender"
      ]
    },
    {
      "id": "O_throw",
      "role": "obligation",
      "source_span": "throw",
      "description": "The function reverts execution",
      "parameters": []
    }
  ],
  "outcomes": [
    "O_throw"
  ],
  "conditions": [
    "C_authorized"
  ],
  "outcome_formula": "O_throw",
  "condition_formula": "not C_authorized"
}
```

Listing 32: Example for parameter **SEMANTIC_ANALYSIS_JSON** in Listing 28.

LLM response | Infer Predicates

```
{
  "actions": [
    {
      "name": "define_rule_vocabulary",
      "input": {
        "sorts": [
          {
            "name": "Address",
            "description": "Ethereum address"
          }
        ],
        "predicates": [
          {
            "atom_id": "C_authorized",
            "name": "AuthorizedCondition",
            "description": "The _from account has deliberately authorized the message sender via some mechanism",
            "parameters": {
              "_from": "Address",
              "sender": "Address"
            }
          },
          {
            "atom_id": "O_throw",
            "name": "ThrowObligation",
            "description": "The function reverts execution",
            "parameters": {}
          }
        ]
      }
    }
  ]
}
```

Listing 33: Example LLM response for *Infer Predicates* using the prompt template in Listing 28.

H.4 FORMALAGENT

H.4.1 INFER DSL

Prompt template | Infer DSL

```
Translate the clause into the minimum DSL statement(s) needed to preserve all governed fragments.
Rules:
- Return exactly one statement ending in `;`, unless the source clause contains multiple explicit normative modalities. In
that case, return one or more statements whose semantic union covers the whole clause and whose individual prefixes preserve
the modality governing each fragment.
- Use every listed predicate wherever the supplied formula places it. A predicate may occur more than once when the formula
itself repeats it.
- Treat every required outcome predicate as a separate semantic coverage obligation. In particular, normal completion/state
behavior and an emitted effect are not interchangeable; when both are listed, both must occur in the DSL.
- Reproduce the supplied outcome and condition formulas exactly, including `and`, `or`, `not`, and grouping.
- Preserve the clause's polarity, boolean relationships, argument roles, and explicit constants.
- Do not invent predicates, arguments, conditions, facts, or concrete values.
- A predicate call is already a truth-valued proposition. Never pass `true` or `false` merely to say that a predicate or
alternative holds; use Boolean literals only for an explicit Boolean participant/value in the clause.
- `TargetAttr("path")` may only reference an exact path present in the supplied JSON.
- `TargetAttr` is declaration metadata, never a runtime return value. Use `TargetReturn()` for the sole/first runtime return
value, or `TargetReturn(index)` for a zero-based return index.
- Predicate signatures below describe positional argument roles and sorts; do not copy role names, sort names, or `:` into a
call.
- A parameter name repeated across predicate signatures is one shared semantic variable. Use exactly the same DSL value at
every occurrence of that name.

DSL syntax:
- `Outcome(arguments);`
- `OutcomeA(arguments) and OutcomeB(arguments) if Condition(arguments);`
- Both required outcomes and conditions may use `and`, `or`, `not`, and parentheses.
- Values may be quoted strings, numbers, `true`, `false`, identifiers, `TargetAttr("path")`, or `TargetReturn()`/`
TargetReturn(index)`.
- Every unquoted identifier must match `[A-Za-z_][A-Za-z0-9_]*`; quote any other literal.
- Quoted strings must use JSON double quotes, never single quotes.
- Never paste a declaration or other source fragment as an unquoted value. Use an exact `TargetAttr` path when the
corresponding value is available in the JSON.
- Never use a return's `.type` or `.name` attribute where a predicate needs its runtime value.

Normative modality: Choose MUST, SHOULD, MAY, or no modality; do not strengthen or weaken the source modality.

Runtime returns: `TargetReturn()` is available for the first return value when the count below is nonzero; use `TargetReturn(
index)` for additional return values.
Runtime return count: <<RETURN_COUNT>>.


Return JSON only, exactly in the shape `{"dsl":"...;"}`.

<<BOUND_RULE_VOCABULARY>>

Clause:```
<<RULE>>
```

Associated target:```
<<TARGET_DECLARATION>>
```

Allowed scalar `TargetAttr` paths and values:```
<<FLATTENED_TARGET_ATTRIBUTES_JSON>>
```
```

Listing 34: The prompt template of *Infer DSL*. *(For parameters involved in the template, **RETURN_COUNT** represents the number of runtime return values, **BOUND_RULE_VOCABULARY** represents the rule-specific predicates and required Boolean formulas, **RULE** represents the original natural-language rule clause, **TARGET_DECLARATION** represents the associated entity declaration, and **FLATTENED_TARGET_ATTRIBUTES_JSON** represents allowed target-attribute paths and their scalar values.)*

**Input** | Return count

```
1
```

Listing 35: Example for parameter **RETURN_COUNT** in Listing 34.

Input | Bound vocabulary

```
Required outcome predicates:
- ThrowObligation(): The function reverts execution

Required outcome formula: `ThrowObligation`

Condition predicates:
- AuthorizedCondition(_from: Address, sender: Address): The _from account has deliberately authorized the message sender via
some mechanism
```
```

Input | Bound vocabulary *(continued)*

```
Required condition formula: `not AuthorizedCondition`
```

Listing 36: Example for parameter **BOUND_RULE_VOCABULARY** in Listing 34.

Input | Rule

```
The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
```

Listing 37: Example for parameter **RULE** in Listing 34.

Input | Target declaration

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Listing 38: Example for parameter **TARGET_DECLARATION** in Listing 34.

Input | Target attributes

```
{"kind":"function","name":"transferFrom","parameters.0.name":"_from","parameters.0.type":"address","parameters.1.name":"_to","parameters.1.type":"address","parameters.2.name":"_value","parameters.2.type":"uint256","returns.0.name":"success","returns.0.type":"bool","visibility":"public"}
```

Listing 39: Example for parameter **FLATTENED_TARGET_ATTRIBUTES_JSON** in Listing 34.

LLM response | Infer DSL

```
{
  "dsl": "SHOULD ThrowObligation() if not AuthorizedCondition(_from, sender);"
}
```

Listing 40: Example LLM response for *Infer DSL* using the prompt template in Listing 34.

H.5 EntityLocalizer Supplement: Non-empty Scope Discovery in ERC-721

This subsection supplements EntityLocalizer's *Infer Scope Regex* example in Section H.1. The ERC-721 document contains named interfaces that enclose member declarations. The excerpt below shows the `ERC721Metadata` interface and one non-empty scope pattern recorded by the GPT-OSS-20B evaluation run. The named capture group identifies the enclosing interface, while the opening and closing patterns delimit its body; thus, `name`, `symbol`, and `tokenURI` can be associated with `ERC721Metadata`.

The result is an excerpt of the saved `hierarchy_patterns` in `target_description` from evaluation run 1; only the metadata-interface entry is shown.

Input | ERC-721 document excerpt

```
interface ERC721Metadata /* is ERC721 */ {
    /// @notice A descriptive name for a collection of NFTs in this contract
    function name() external view returns (string _name);

    /// @notice An abbreviated name for NFTs in this contract
    function symbol() external view returns (string _symbol);

    /// @notice A distinct Uniform Resource Identifier (URI) for a given asset.
    /// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in RFC
    ///  3986. The URI may point to a JSON file that conforms to the "ERC721
    ///  Metadata JSON Schema".
    function tokenURI(uint256 _tokenId) external view returns (string);
}
```

Listing 41: Example for parameter **DOCUMENT_CHUNK** in Listing 1: an ERC-721 excerpt containing a named lexical scope and three member declarations.

Recorded result | ERC-721 scope pattern

```
{
  "hierarchy_patterns": [
    {
      "kind": "interface",
      "pattern": "^interface\\s+(?P<name>\\w+)\\b.*\\{",
      "example": "interface ERC721Metadata /* is ERC721 */ {",
      "open_pattern": "\\{",
      "close_pattern": "\\}",
      "role": "semantic_container"
    }
  ]
}
```

Listing 42: One inferred scope pattern retained in the ERC-721 evaluation result.

H.6 Baseline Prompts

H.6.1 Spec-in-One (SiO)

Prompt template | Spec-in-One

```
Create a formal specification directly from the supplied document.

Use only the document. Do not assume a programming language, API, standard, domain,
or facts not stated in it.

In this single response:
1. Extract every documented programming entity needed as a rule target, including its exact declaration, source-line range,
lexical hierarchy, and declaration attributes.
2. Extract every independently normative clause and associate it with one extracted entity.
3. Define the logical predicates needed to express those clauses.
4. Formalize every extracted clause as exactly one DSL statement.

Entity contract:
- `text` is the exact verbatim declaration substring from the document; do not include comments, prose headings, or enclosing
braces.
- `start_line` and `end_line` are one-based inclusive physical-file lines covering that declaration. Count an initial blank
physical line as line 1.
- `hierarchy` contains only canonical names of lexical enclosing declarations, outermost first (for example `["ERC721"]`, not
 the full `interface ERC721 ...` declaration). Headings and prose are never hierarchy. Rust receiver templates such as `*const
 T`, `*mut T`, and `Self` belong in `attributes.receiver_template`, never in hierarchy.
- Preserve complete declaration attributes. For Solidity functions include kind, name, ordered parameters, ordered named
returns (preserve both return names and types), visibility, and state_mutability (`nonpayable` when omitted); for events
preserve ordered fields and indexed flags; for interfaces preserve declared inheritance. For Rust methods include kind, name,
receiver, ordered parameters, return_type/returns, visibility, const/unsafe modifiers, method generics/lifetimes/bounds, where
 bounds, and receiver_template. Put lifetime/type/const template variables in attributes, not hierarchy.
- Every formalization `target` must equal exactly one `entities[].text` value.

Extraction contract:
- Copy `rule` text from the document without paraphrasing.
- Copy the associated `target` text without paraphrasing.
- Do not merge independent clauses or return duplicate clauses.

Return JSON only with exactly this shape:
{
  "entities": [
    {
      "text": "verbatim declaration",
      "start_line": 1,
      "end_line": 1,
      "hierarchy": [],
      "attributes": {"kind": "function", "name": "example"}
    }
  ],
  "formalizations": [
    {"rule": "verbatim clause", "target": "verbatim target", "dsl": "OneStatement(...);"}
  ]
}

Document:```
<<DOCUMENT>>
```
```

Listing 43: SiO: direct specification generation in a single response. ***(DOCUMENT**: the complete API document, preserving physical source lines.)*
```

H.6.2 GRAMMAR-IN-ONE (GIO)

Prompt template | Grammar-in-One

```
In this single response, extract all rule-target entities, extract every independently
normative rule, and design one global logical DSL grammar covering all extracted rules.
Do not formalize any rule and do not return DSL statements. A separate FormalAgent will
use your grammar only after it passes local compilation. The detected language is
<<LANGUAGE>>; use only facts present in the document.

Entity contract:
- `text` is the exact verbatim declaration substring from the document, excluding comments, prose headings, and enclosing
braces.
- `start_line` and `end_line` are one-based inclusive physical-file lines.
- `hierarchy` contains only canonical names of lexical enclosing declarations, outermost first. Headings, prose, Rust receiver
 templates (`*const T`, `*mut T`, `Self`), generic variables, and lifetime variables are never hierarchy.
- Preserve complete declaration attributes. For Solidity functions include kind, name, ordered parameters, ordered named
returns with both name and type, visibility, and state mutability; for events preserve ordered fields and indexed flags; for
interfaces preserve inheritance. For Rust methods include kind, name, receiver, ordered parameters, return information,
visibility, const/unsafe modifiers, method generics/lifetimes/bounds, where bounds, and `receiver_template`.

Rule contract:
- Copy `rule` verbatim from the document; do not paraphrase, merge independent clauses, duplicate clauses, promote examples/
rationale/headings to rules, or use outside knowledge.
- Every `target` must exactly equal one returned `entities[].text` declaration.
- Preserve every independently observable result, state transition, effect, obligation, prohibition, permission, precondition,
 failure condition, exception, and modality.

Global grammar contract:
- Return one self-contained Lark grammar shared by every extracted rule. It must compile using `Lark(grammar, start="start",
parser="lalr")`.
- The start rule must accept one or more `stmt` productions. A statement supports an optional modality (MUST, SHOULD, or MAY),
 a Boolean outcome expression, an optional `if` Boolean condition, and a terminating semicolon. Preserve these modalities
explicitly.
- Support `and`, `or`, `not`, parentheses, positional values, `TargetAttr("path")`, `TargetReturn()`, and `TargetReturn(index
)`.
- Give outcome predicates productions named `primary_<identifier>` and condition predicates productions named `cond_<
identifier>`. Bind each callable predicate name as a quoted literal directly in its semantic production and put a non-empty
`//` semantic description on every such production.
- Include all imports, tokens, punctuation, value productions, dispatch rules, and predicates needed for the complete rule
inventory. Do not include a DSL formalization in the response.

Return JSON only with exactly this shape:
{
  "entities": [
    {
      "text": "verbatim declaration",
      "start_line": 1,
      "end_line": 1,
      "hierarchy": [],
      "attributes": {"kind": "function", "name": "example"}
    }
  ],
  "rules": [
    {"rule": "verbatim normative clause", "target": "verbatim declaration"}
  ],
  "grammar": "complete global Lark grammar"
}

Document:```
<<DOCUMENT>>
```
```

Listing 44: GiO: joint entity and rule extraction with global grammar generation. *(**DOCUMENT**: the complete API document; **LANGUAGE**: the language hint inferred from the source path.)*
```