FROM JUDGE TO VERIFIER: FORMALLY VERIFIED FAITHFULNESS FOR EVIDENCE-GROUNDED GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-as-a-Judge offers no independent trust boundary: the same class of model that may introduce unsupported claims is trusted to detect them. We study claim-level grounding as a checkable criterion: every atomic claim in a generated artifact must be entailed by its upstream evidence. We introduce VERLEAN, a neurosymbolic verifier in which an LLM translates evidence and claims into a finite-domain model and entailment queries, and a deterministic Lean 4 kernel decides each formalized query. We develop it on a proof-of-concept regulatory-compliance dataset with two transformation tasks (1,624 claims: *fact-to-summary* and *summary-to-requirement*), test generalization on 444 public legal summaries, and stress-test on a synthetic benchmark of typed errors from public contracts (5,550 judgments). Across the three settings with a direct LLM-as-a-Judge comparison, VERLEAN raises F1 (0.747 to 0.917 *fact-to-summary*, 0.752 to 0.894 *summary-to-requirement*, 0.806 to 0.943 legal summaries), and on the typed benchmark detects unsupported claims at 0.895 recall and 0.935 F1. Its residual misses trace to neural translation, not the kernel; an ablation attributes the *accuracy* to the formalization, not the evaluator that reads it. The grounding decision is then a settled, mechanical check. The kernel decides it with no further LLM call, returning identical verdicts across runs. It thus dominates an LLM evaluator, which only matches that accuracy while adding cost and non-reproducible variance (LLM judges disagree on up to 44% of the subtlest errors). Failed queries yield counterexamples that repair more outputs, and sooner, than free-form judge rationales (first-attempt recovery up 4.3–12.9 points, reaching 90.9–97.0% in six attempts). Grounding decisions become reproducible, auditable, and, through their counterexamples, repairable.

1 INTRODUCTION

Large language models (LLMs) increasingly produce not just text but artifacts that drive real decisions, so their value hinges less on fluency than on faithfulness: whether every claim is grounded in its source evidence. This is acute in regulatory compliance, where translating dense regulatory obligations into product requirements means that dropping a condition, broadening a scope, or changing a modality can silently alter a compliance decision. LLMs automate such transformations in software and requirements engineering workflows (Hou et al., 2024; Norheim et al., 2024), and because these workflows are multi-stage, an unsupported intermediate claim can propagate into downstream artifacts and system behavior. We therefore study claim-level grounding across successive transformations: whether each generated claim is semantically entailed by its upstream evidence, decided deterministically rather than by probabilistic judgment.

A common evaluation strategy uses another LLM as a judge to assess whether a claim is supported by its evidence (Zheng et al., 2023; Bavaresco et al., 2025). However, LLM-judge decisions are probabilistic and vary across models and tasks: their accuracy hinges on the capability of the judge model itself, and even strong judges exhibit systematic biases (Chen et al., 2024) and disagree with one another, most of all on the subtle violations that matter here, such as dropped conditions and modality shifts. They therefore provide no independently checkable entailment decision. This motivates our central research question: *How can we verify grounding across multiple transformations in a high-stakes LLM pipeline without entrusting the final entailment decision to another LLM?*

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.

We introduce VERLEAN, a neurosymbolic verifier built on the principle that *the LLM translates, while a deterministic kernel decides*: an LLM maps evidence to a finite-domain semantic model and each atomic claim to condition–conclusion queries, and a Lean 4 (Moura & Ullrich, 2021) checker tests whether each conclusion holds in every world satisfying its condition, returning counterexamples that guide regeneration. To our knowledge, VERLEAN is the first to verify claim-level grounding of regulatory artifacts with a deterministic theorem-prover kernel rather than a learned judge. The kernel is exact about the formal model the LLM builds, but not about whether that model faithfully captures the evidence, so we evaluate the end-to-end verdict against human annotations on three testbeds (Section 5): two grounding transformations in a compliance workflow (1,624 claims), public legal summaries, and a larger synthetic benchmark of typed errors. Across all three, formalization raises recall and F1 over direct LLM judging while preserving high precision; the kernel adds deterministic, auditable verdicts and counterexamples that improve repair over free-form rationales.

Our contributions are:

- **Formulation.** We cast grounding as entailment from upstream evidence to atomic claims, independent of the transformation that produced them and checkable at each stage.
- **Method.** VERLEAN, a neurosymbolic verifier that hands semantic translation to an LLM but decides the entailment check with a deterministic engine (Lean 4), not a second LLM.
- **Detection.** Formalizing evidence and claims lifts recall while keeping precision high across two transformations and a public dataset; an ablation traces the gain to the formalization, and the residual misses to translation, not the kernel.
- **Repair.** Its counterexamples turn each failure into a constraint that guides regeneration into grounded revisions, more effectively than free-form LLM-judge rationales.

2 BACKGROUND

Faithfulness in evidence-grounded generation is a claim-level property: every atomic assertion an artifact makes should be *entailed* by the evidence it was generated from. A claim can be fluent, true, well-sourced, or on-topic and still fail to be grounded, for instance by dropping a qualifier or asserting more than the evidence allows. Grounding asks only whether the claim follows from its evidence; correctness, plausibility, relevance, and provenance are neither sufficient nor the target.

Regulatory compliance is a setting where this property is both consequential and checkable. Organizations translate regulatory obligations into product requirements, and each claim must trace back to explicit source evidence. The translation runs in chained stages, and the errors that matter are small and local: dropping an applicability condition, broadening a scope or quantifier, strengthening a modality, or swapping an entity or role can each silently alter a compliance decision, and an early error propagates into every artifact derived from it. These properties of high stakes, explicit provenance, and staged transformation make compliance a natural proof-of-concept setting, and we build a dataset from one such workflow to develop and first test VERLEAN (Section 5.1).

3 PROBLEM FORMULATION

Let $E = \{e_1, \ldots, e_m\}$ be the evidence for a generated artifact A, which is decomposed into atomic claims $C(A) = \{c_1, \ldots, c_n\}$. A claim is VALID if $E \models c_i$ (i.e., E entails c_i); otherwise, it is NOT_VALID. The latter includes contradictions, unsupported additions, entity or role substitutions, and semantic overreach caused by dropped conditions, strengthened quantifiers, or stronger modalities. An artifact is valid exactly when all its claims are entailed: $y(E, A) = \text{VALID} \iff \forall c_i \in C(A),\ E \models c_i$.

VERLEAN separates semantic translation from entailment checking. The *checker* is VERLEAN's verification stage that enumerates worlds and evaluates queries, and the trusted Lean 4 *kernel* deterministically decides each query. An LLM maps E to a finite-domain semantic model M_E and each claim c_i to condition–conclusion queries $Q(c_i) = \{(\phi_{ij}, \psi_{ij})\}_{j=1}^{k_i}$.

Concretely, VERLEAN encodes $M_E = (\texttt{World}, \mathcal{P})$ directly in Lean 4. The type $\texttt{World} = D_1 \times \cdots \times D_n$ is a Lean 4 `structure` whose fields are finite domains D_i (`Bool` or enumerated `inductive` types), one per condition dimension the evidence distinguishes, so $W(M_E) = \texttt{World}$

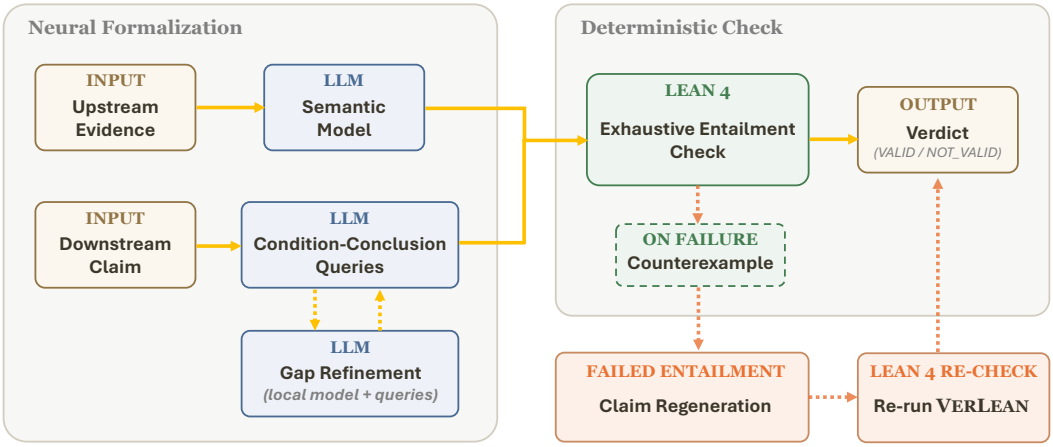

Figure 1: **Overview of VERLEAN.** The neural stage translates evidence and each claim into a formal model and condition–conclusion queries; the Lean 4 kernel decides them and, on failure, the checker returns a counterexample that drives constrained regeneration. The LLM proposes; the kernel decides.

and $|W(M_E)| = \prod_i |D_i|$. Each *policy function* in $\mathcal{P}$ is a total Lean 4 `def` g : `World` $\to \tau_g$ mapping a world to an outcome the evidence allows (an obligation, a permitted value, a category), and every query condition ϕ_{ij} and conclusion ψ_{ij} is a Lean 4 Boolean `def` over `World` built from these policy functions. Because the fields are finite and the functions total, the kernel decides each query by evaluation rather than by a learned judgment; the LLM only instantiates the fields, functions, and query bodies per artifact (Section 4; see Appendix D for a worked `World`/`def` encoding). The checker accepts c_i only if every query holds in every world:

$$\forall j,\ \forall w \in W(M_E), \quad \phi_{ij}(w) \Rightarrow \psi_{ij}(w).$$

This grounding check assumes the condition is satisfiable, i.e., $\exists w \in W(M_E) : \phi_{ij}(w)$; if not, the implication holds vacuously and the claim passes untested. The checker does not detect this, so we treat a vacuous pass as a translation failure, not valid grounding.

A failed query yields a counterexample satisfying $\phi_{ij}(w) \wedge \neg\psi_{ij}(w)$. Operationally, VERLEAN tests $M_E \models Q(c_i)$ by finite-domain model checking over a single autoformalized model, rather than deciding the open-world entailment $E \models c_i$: where the evidence is silent, the total functions fall back to sentinel defaults (Appendix A.2), a deliberate closed-world convention. The check is deterministic once the formalization is fixed, but end-to-end validity depends on whether the LLM faithfully represents the evidence and claims.

4 VERLEAN: A NEUROSYMBOLIC GROUNDING VERIFIER

4.1 OVERVIEW

VERLEAN implements the trust boundary from Section 3 in four stages (Figure 1): (1) a structure-aware splitter decomposes the artifact into atomic claims; (2) an LLM encodes the evidence as a finite-domain semantic model; (3) the LLM translates each claim into condition–conclusion queries that Lean 4 checks against the model; and (4) a coverage audit re-verifies assertion content omitted from those queries against the source. LLMs propose formalizations and evidence mappings, and the kernel decides every formalized entailment query. A claim is accepted only if all checks pass; a rejection comes from a kernel counterexample or an unresolved gap, and the checker turns failures into repair signals for constrained regeneration.

4.2 NEURAL FORMALIZATION: THE SEMANTIC MODEL AND CLAIM QUERIES

The input is the evidence for an artifact: the source statements it must be grounded in, whatever their form (e.g., regulatory source facts, prior-stage summaries, or a source document).

VERLEAN builds the semantic model in two steps:

1. **Condition dimensions.** The LLM extracts the finite dimensions the evidence distinguishes: Boolean and enumerated fields for entity roles, transaction types, and applicability conditions. Together these define a typed `World`.
2. **Policy functions.** Each source statement becomes one or more typed functions over `World` encoding the outcomes it allows: obligations, permissions, prohibitions, numeric values, categorical results. Translation stays within the supplied evidence and adds no external knowledge.

Because every field is finite, VERLEAN enumerates their Cartesian product. Cross-field dependencies must be encoded explicitly; otherwise the checker treats all combinations as possible, including impossible worlds that may yield spurious counterexamples.

The LLM then translates each claim into a set of condition–conclusion queries: a condition specifying when an assertion applies and a conclusion expressing the asserted outcome using the source-grounded functions. Multiple assertions within one claim yield multiple queries. A *coverage audit* flags assertions the queries did not represent; for each, *LLM gap refinement* (Figure 1) searches only the evidence for a supporting statement and, if found, builds a self-contained local model so the kernel can re-verify it, and otherwise leaves it unsupported. Any assertion left without a source-grounded formalization is conservatively rejected.

4.3 DETERMINISTIC CHECK

For each formalized query, Lean 4 decides whether the conclusion holds in *every* world satisfying the condition: the query reduces to a closed decidable Boolean, kernel-checkable via `decide` (D.1), not a learned judgment. When the proposition fails, the checker separately enumerates the finite world space to extract a witness; this diagnostic search lies outside the trusted decision boundary. The check is therefore sound and complete *relative to the formal model*. A claim is accepted only when all its queries pass; if any fails, VERLEAN returns NOT_VALID with the counterexample, pinpointing where the claim exceeds its evidence. With no LLM in the decision and the formalization fixed, the same input always yields the same verdict.

Why Lean 4? Determinism alone does not require an interactive theorem prover, since SMT or FOL solvers can perform the same finite checks and may scale or extract countermodels better. We use Lean as an auditable substrate: its small kernel checks typed, self-contained, re-runnable artifacts, its representation extends to richer specifications, and it is increasingly accessible to LLMs through autoformalization work (Azerbayev et al., 2023; Ying et al., 2024; Jiang et al., 2024). Symbolic solvers remain complementary for larger state spaces.

4.4 ON FAILURE: COUNTEREXAMPLE-GUIDED REGENERATION

VERLEAN turns the failing counterexample into a natural-language repair signal: the offending world and the field values under which the conclusion failed. It hands this to an LLM with the artifact and evidence, so the rewriter targets exactly where the claim over-reaches rather than a bare verdict. To avoid new drift, regeneration is restricted to two evidence-bounded moves: *narrow*, restoring a condition or weaker modality the evidence supports, and *delete*, removing an unsupportable assertion; it adds no new information. VERLEAN re-runs the pipeline on the revision, and a repair succeeds only when the content is narrowed or removed and re-verification passes (A.6).

5 EVALUATION

We evaluate whether VERLEAN's neural–symbolic trust boundary yields more reliable and actionable grounding verification than LLM judging, via four research questions:

RQ1: How effectively does VERLEAN detect unsupported claims across the two evidence-to-artifact transformations compared with LLM judging?

RQ2: Does VERLEAN generalize beyond the compliance setting to legal summaries?

RQ3: Can VERLEAN's counterexamples guide regeneration of artifacts into grounded revisions?

Table 1: Evaluation datasets and gold labels (count, within-dataset %). High NOT_VALID rates reflect our strict per-assertion entailment criterion, not poor data quality.

Dataset	Unit	VALID	NOT_VALID	Total
Fact-to-summary	Claim	430 (62.7%)	256 (37.3%)	686
Summary-to-requirement	Claim	592 (63.1%)	346 (36.9%)	938
Legal summaries	Summary	126 (28.4%)	318 (71.6%)	444
Synthetic contracts (CUAD)	Claim	3,475 (62.6%)	2,075 (37.4%)	5,550

RQ4: At larger scale, which categories of grounding error does VERLEAN detect well, and where does its neural translation limit detection?

5.1 COMPLIANCE WORKFLOW AND GROUNDING TASKS

Our primary testbed is a multi-stage regulatory-compliance workflow: source facts are condensed into structured summaries, and those summaries, with other evidence, into downstream requirements. We study two claim-level grounding tasks along this chain. The *fact-to-summary* task checks whether each summary claim is entailed by its linked source facts; the *summary-to-requirement* task checks whether each requirement claim is entailed by the joint evidence of its linked summaries, *system facts* (platform and system behavior), and *technical decisions* (human-made engineering choices). We verify grounding at each stage against the evidence feeding it, calling a claim not entailed by its evidence NOT_VALID (the positive class). We split each artifact into atomic claims with structure-aware rules (sections, list items, sentences) and check each independently.

5.2 EVALUATION DATA AND GOLD LABELS

We use four settings (Table 1), profiled in Appendix B. Two are the *fact-to-summary* and *summary-to-requirement* tasks of Section 5.1 (the *compliance* data), keeping the 1,624 claims with evidence provenance. To test transfer beyond this schema, we add 444 source–summary pairs from the public *Plain English Summarization of Contracts* dataset (Manor & Li, 2019); each summary is a single claim-length unit, counted VALID only if its source entails it. Because neither dataset labels errors by type or reaches large scale, we build a controlled synthetic benchmark from public commercial contracts (Section 5.7): the same detection task with every injected violation labeled by type.

Initial labels came from a majority vote of three models (Opus 4.8, Sonnet 4.6, GPT-5.6-sol). Roughly half of each dataset underwent human review, covering all disagreements, prompt-defined low-confidence and high-risk cases, and a random audit of unanimous decisions, and labels were frozen before evaluation.

5.3 SYSTEMS AND EVALUATION PROTOCOL

We compare VERLEAN with a direct LLM-as-a-Judge baseline. Atomic-claim splitting is a shared preprocessing step run once before either system: the compliance tasks are already per claim, and each public summary is a single claim-length unit, so both systems judge the same claim against the same evidence with the same model, Opus 4.8. The baseline is therefore a *decompose-then-judge* baseline: it reads the atomic claim and returns one VALID/NOT_VALID verdict in natural language. We keep it to this standard single-pass protocol, with no ensembling or chain-of-thought prompting, so that the use of formalization is the only difference between the two systems. Adding such scaffolds to the judge would raise its own compute and mix that effect with the effect of formalization. VERLEAN instead formalizes the same claim and evidence and delegates the verdict to the Lean 4 kernel, accepting a claim only when all its entailment and coverage checks pass. Because both see identical atomic claims, the comparison isolates the formal check as the sole difference between them, so any gain is attributable to formalization and not to finer decomposition. To then separate formalization from the kernel itself, we also evaluate a *Formalize-then-judge* ablation that runs VERLEAN's full formalization pipeline but lets the same LLM, not the kernel, evaluate the resulting model and queries, holding pipeline compute fixed and changing only the decision rule (Section 5.8). We treat NOT_VALID as the positive class and report precision, recall, F1, and accuracy, emphasizing recall and F1 because a missed unsupported claim propagates silently downstream while a flagged one is only sent for review.

Table 2: Compliance claim-level grounding results (%). Unsupported claims are the positive class; bold marks the better method.

Task	Method	Precision	Recall	F1	Accuracy
Fact-to-summary	LLM Judge	**97.5**	60.5	74.7	84.7
	VERLEAN	92.8	**90.6**	**91.7**	**93.9**
Summary-to-requirement	LLM Judge	77.5	73.0	75.2	82.5
	VERLEAN	**97.0**	**82.9**	**89.4**	**92.8**

Table 3: Grounding performance on legal summaries (%).

Method	Precision	Recall	F1	Accuracy
LLM Judge	97.8	68.6	80.6	76.4
VERLEAN	**98.0**	**90.9**	**94.3**	**92.1**

5.4 COMPLIANCE GROUNDING RESULTS

Addressing RQ1, we report detection accuracy on the two transformations, from single-source facts to increasingly varied evidence (Table 2).

Fact-to-summary grounding. VERLEAN achieves higher F1 (Table 2): recall rises sharply while precision dips only slightly, so it catches far more unsupported claims at little cost.

Summary-to-requirement grounding. VERLEAN likewise achieves higher F1 under varied evidence (Table 2), and here precision rises rather than falls. Distributing evidence across multiple sources makes the baseline judge more error-prone, whereas explicitly formalizing each source lets VERLEAN improve every metric, so the formulation helps more as claims draw on multiple evidence types.

Cross-task comparison. VERLEAN improves F1 on both transformations (RQ1). Its recall on *summary-to-requirement* nonetheless falls below the *fact-to-summary* level: varied, multi-source evidence must be combined into a single consistent `World`, raising the formalization burden and the risk of an omitted or misaligned qualifier.

5.5 GENERALIZATION TO LEGAL SUMMARIES

To test whether VERLEAN generalizes beyond the compliance setting (RQ2), we apply it to an external, public dataset sharing neither its schema nor its provenance: legal document excerpts (contract or terms-of-service text) paired with human-written plain-English summaries. We choose it precisely because it aims at readable simplification rather than strict entailment. Its summaries may abstract, compress, or simplify the source (Manor & Li, 2019), so many fail our per-assertion criterion by adding information, mismatching scope, or dropping conditions and qualifiers. These naturally occurring violations give a testbed of grounding failures under realistic distribution shift.

VERLEAN preserves the baseline's high precision, 98.0%, while sharply raising recall to 90.9% (Table 3). Since NOT_VALID is the majority class here, at 71.6%, high recall on its own would be unremarkable: a detector that flagged every summary would reach it while destroying precision on the 126 faithful summaries. VERLEAN instead holds precision at 98.0% while raising recall, so it tells faithful simplifications apart from ungrounded ones rather than penalizing all abstraction. This answers RQ2 positively: VERLEAN transfers beyond the compliance setting to a dataset with a different schema and provenance.

5.6 FROM DETECTION TO REPAIR: COUNTEREXAMPLE-GUIDED REGENERATION

To isolate repair quality from detection (RQ3), we use only outputs both methods correctly reject (143 *fact-to-summary*, 201 *summary-to-requirement*, 209 legal summaries); both share evidence, rewriter, narrow-or-delete constraints, and verification, differing only in the guidance, over up to six attempts (@k: % passing re-verification within k).

Table 4: Cumulative repair recovery on jointly detected unsupported outputs. @k denotes recovery within k attempts (%).

Dataset	Repair guidance	@1	@2	@3	@4	@5	@6
Fact-to-summary	LLM-Judge rationale	60.8	80.4	84.6	86.7	89.5	90.9
	VERLEAN counterexample	**67.8**	**87.4**	**93.0**	**95.1**	**95.8**	**95.8**
Summary-to-requirement	LLM-Judge rationale	66.7	83.6	90.0	93.5	96.0	**97.0**
	VERLEAN counterexample	**79.6**	**95.0**	**96.5**	**97.0**	**97.0**	**97.0**
Legal summaries	LLM-Judge rationale	65.1	75.1	80.9	84.2	86.1	88.5
	VERLEAN counterexample	**69.4**	**80.4**	**86.1**	**88.0**	**90.4**	**90.9**

Table 5: VERLEAN detection and one-round repair on the synthetic contract benchmark. Recall is the percentage of injected errors of each type detected (NOT_VALID is the positive class); Injected and Detected are counts. Repair@1 is the percentage of *detected* errors re-verified as grounded after a single counterexample-guided rewrite. Judge disagreement is how often the three LLM judges (Opus 4.8, Sonnet 4.6, GPT-5.6-sol) disagree on a claim (VERLEAN is deterministic, so 0). Disagreement is over injected errors, higher (pooled 10.0%) than the 7.2% over all 5,550 claims.

Type	Error category	Injected	Detected	Recall (%)	Repair@1 (%)	Judge disagreement (%)
T1	Added information	300	299	99.7	98.3	0.7
T2	Dropped condition/qualifier	299	196	65.6	91.8	44.1
T3	Over-broad quantifier	300	282	94.0	87.9	6.7
T4	Modality shift	281	216	76.9	90.3	16.0
T5	Altered numeric value	296	288	97.3	86.5	0.0
T6	Entity/actor swap	299	295	98.7	86.1	2.3
T7	Direct contradiction	300	282	94.0	87.6	0.3
Pooled (T1–T7)		2,075	1,858	89.5	89.7	10.0

Table 4 answers RQ3 positively: VERLEAN's counterexamples guide both faster and more complete repair than LLM-Judge rationales. It improves first-attempt recovery by 4.3–12.9 points and reaches 97.0% on *summary-to-requirement* in four attempts, not six. Its gains persist at $k = 6$ on the other two tasks: it repairs more outputs within a fixed budget, not the same ones sooner.

Repairs do not always succeed at once: a rewrite may under-narrow or add a new defect, so every revision is rechecked and each new counterexample constrains the next. Both conditions are scored by the same VERLEAN re-verification, validated against human labels above, so recovery uses an externally benchmarked criterion, not a bare internal fixpoint; a spot-check found accepted repairs to be faithful narrowings, though scoring both with VERLEAN may still favor its own guidance.

5.7 SCALING AND ERROR-TYPE ANALYSIS ON SYNTHETIC CONTRACTS

The legal summaries are small and untyped, so they cannot show *which* grounding errors VERLEAN catches. For a larger, typed benchmark (RQ4), we build a controlled synthetic set from the public Contract Understanding Atticus Dataset (CUAD) (Hendrycks et al., 2021) of commercial contracts.

Benchmark construction. From 300 CUAD contracts we take source excerpts comparable in scale to the compliance facts and generate near-extractive atomic-claim summaries. Each claim is certified as grounded under the same protocol as above, forming the VALID gold. We then inject one error per variant, a minimal single-point edit that realizes one of the seven violation types seen in the compliance setting (T1–T7 in Table 5), so the injected claim is NOT_VALID by construction with a known type. The result is 5,550 units (3,475 grounded, 2,075 injected), an order of magnitude larger; gold labels are set without VERLEAN, so the comparison is independent.

Detection and error-type breakdown. Pooled over the 2,075 injected errors, VERLEAN reaches 89.5% recall at 97.7% precision (F1 93.5%), wrongly flagging only 1.2% of grounded claims. The per-type breakdown (Table 5) answers RQ4 and splits the errors into two regimes. VERLEAN is

near-perfect on errors that contradict a value the model encodes (T1, T3, T5, T6, T7), each of which yields a counterexample world. Recall is lower on errors that drop a qualifier or strengthen modality (T2, T4), because detection requires both the relevant distinction and a separating world. These are end-to-end VERLEAN failures arising in neural formalization, not kernel evaluation, and mark its current recall ceiling on these two types. Because such conditions and modalities carry the most weight in high-stakes settings, typed scope and modality representations and round-trip checks are promising mitigations, with targeted human review until then.

Typed repair (RQ3 at scale). A single repair round (vs. up to six in Section 5.6) on the 1,858 detected errors leads 89.7% to pass re-verification (Repair@1 in Table 5), almost entirely by narrowing rather than deleting (1,640 vs. 27). Two findings stand out. First, the least-detected types (T2, T4) are among the most reliably repaired once detected, because the counterexample names the exact missing qualifier or modality: detection, not repair, is their bottleneck. Second, repair rarely trades one error for another: an independent Opus 4.8 classifier flags only 19 of 1,858 rewrites (1.0%) as a different error type, and regressions usually over-narrow into a new dropped condition.

LLM judges disagree where formalization is hardest. The typed labels also quantify how *inconsistent* LLM judging is. Across the three judges on all 5,550 claims, disagreement (Table 5) is 7.2% overall but highly uneven: near zero on commission errors (T1, T5, T7) and concentrated on the subtle types (T2 at 44.1%, T4), and even the two strongest judges disagree on 24% of T2. These are exactly where VERLEAN's formalization is hardest and a single "correct" label is least well defined, yet the kernel returns the same verdict on every run, so the deterministic boundary is most valuable where LLM judging is least trustworthy.

5.8 ABLATION: FORMALIZATION OR THE KERNEL?

Does the gain over direct judging come from the formalization or from the kernel that decides it? *Formalize-then-judge* separates the two: it runs VERLEAN's full formalization pipeline (same `World`, policy functions, and queries) and changes only the evaluator, asking the same LLM rather than the kernel to read the generated code and return each verdict. Only the decision rule differs.

It tracks VERLEAN's detection F1 within 0.6 points on every setting, far above the LLM judge: 91.8 vs. 91.7 on *fact-to-summary*, 88.8 vs. 89.4 on *summary-to-requirement*, and an identical 94.3 on legal summaries (LLM judge 74.7, 75.2, 80.6). The accuracy is therefore built into the formal structure, not the evaluator: even an imperfect LLM reading the model recovers nearly all of it. This licenses the kernel rather than making it redundant: once the formalization fixes the answer, deciding a query is a settled, mechanical check, and the evaluators are accuracy-equivalent by construction, so the choice is about cost and trust, not accuracy; there the kernel dominates. Formalize-then-judge pays an extra LLM call per verdict to re-derive what the model already determines, stays model-dependent, and yields no counterexample, whereas the kernel returns the identical verdict every run, removing the 7.2–44% LLM-judge disagreement of Table 5, and emits the counterexample that drives repair (Section 5.6). Formalization buys the accuracy; the kernel then decides deterministically, with no model call, and auditably. This comparison is compute-matched: only the evaluator differs.

6 RELATED WORK

Claim-level grounding and factuality evaluation. Prior work evaluates source support with factual-consistency classifiers and natural-language inference (Kryściński et al., 2020; Laban et al., 2022). FActScore decomposes generations into atomic claims and assesses each against retrieved evidence (Min et al., 2023). LLM-based evaluators are more flexible, but their agreement with humans varies across tasks and models, with systematic biases (Zheng et al., 2023; Liu et al., 2023; Bavaresco et al., 2025; Chen et al., 2024). VERLEAN adopts the atomic-claim view but replaces the learned decision with explicit entailment queries decided by a deterministic Lean 4 kernel.

Neurosymbolic verification and autoformalization. Logic-LM and LINC translate language into logic for external solvers or provers (Pan et al., 2023; Olausson et al., 2023); other work rejects answers inconsistent with a prover-checked formalization (Zhou et al., 2024). These separate neural translation from symbolic inference, but solver correctness cannot ensure linguistic fidelity,

motivating semantic evaluation, round-trip verification, and repair (Lahiri, 2024; Amrollahi et al., 2026). VERLEAN applies this separation to evidence-grounded generation, formalizing evidence and claims, checking entailment, and returning counterexample worlds for regeneration.

Lean and language-to-Lean autoformalization. Lean 4's small kernel checks generated proof terms (Moura & Ullrich, 2021), and recent work uses LLMs to autoformalize mathematics, logical reasoning, and software requirements in Lean (Azerbayev et al., 2023; Ying et al., 2024; Jiang et al., 2024; Ye et al., 2026). Elaboration catches malformed expressions and hallucinated definitions and kernel checking rejects invalid derivations, but neither ensures fidelity to the original language (Liu et al., 2025).

Legal formalization and agentic requirements engineering. Formal methods have long represented legislation, contracts, and requirements in machine-checkable languages. Catala and Symboleo structure legal rules and contractual obligations, and ContractCheck translates contract models into first-order constraints for SMT-based consistency checking (Merigoux et al., 2021; Parvizimosaed et al., 2022; Khoja et al., 2022). Work on deontic formalization distinguishes obligations, permissions, prohibitions, exceptions, and defeasible rules (Horner et al., 2025).

In parallel, LLM systems generate software specifications from stakeholder input: MARE coordinates agents for requirements elicitation, modeling, verification, and specification, while SRS systems turn unstructured language into traceable requirements (Jin et al., 2024; Krishna et al., 2024; Zhu et al., 2025). Spec-driven coding agents use generated requirements to guide design and code generation; Kiro autoformalizes requirements as SMT constraints and checks ambiguity, inconsistency, and incompleteness before implementation (Kiro, 2026), since underspecified descriptions can yield executable but semantically incorrect code (Larbi et al., 2025; Yang et al., 2025).

These systems establish specifications as control artifacts for agentic development, but assess their internal quality or alignment with user intent, not provenance. VERLEAN instead asks whether each generated summary or requirement is entailed by the upstream evidence it was derived from, verifying evidence-to-artifact transformations at each stage.

7 DISCUSSION AND LIMITATIONS

Implications. VERLEAN moves neural uncertainty rather than removing it: the LLM carries the semantic translation, but once it is fixed the grounding decision is a settled check the kernel runs deterministically, with no model call and a re-runnable audit trail. This puts the trust boundary in an explicit model a human can inspect rather than a hidden judgment, and matters most where verdicts must be reproducible and their failures explainable.

Limitations. VERLEAN's guarantees are relative to its formalization: mistranslations, omissions, or missing dependencies can yield wrong verdicts. Boolean obligations conflate `false` with `unspecified` and fixed return types collapse deontic distinctions, leaving typed modalities to future work. Exhaustive checking costs $\prod_i |D_i|$, limiting current use to localized models; larger systems may need symbolic solving. The compliance study is small-scale, public labels carry AI-assisted annotation risk, and results use only Opus 4.8. To isolate formalization, our judge is single-pass, not a compute-matched chain-of-thought or self-consistency judge, a natural extension for future work. Grounding checks upstream consistency, not external correctness.

8 CONCLUSION

We introduced VERLEAN, a neurosymbolic grounding verifier that gives semantic translation to an LLM and the entailment decision to a deterministic Lean 4 kernel. Across two compliance transformations and public legal summaries it reaches higher F1 than direct LLM judging, and a typed benchmark traces its residual misses to the neural translation, not the kernel. An ablation attributes the accuracy to formalization; the check is then settled, so running it with the kernel rather than a second LLM is a strict win: the same verdict at no added cost, reproducibly, with a counterexample for repair. This trust boundary makes grounding verification for multi-stage generation more reliable and auditable.

AI USE STATEMENT

We used generative AI tools in three ways. *Method and baseline:* LLMs are core components of the studied system: VERLEAN uses them for the semantic translation of Section 4, and the LLM-as-a-Judge baseline is itself an LLM. *Data:* we used LLMs to generate the synthetic benchmark of Section 5.7 (near-extractive summaries and typed error injections) and to produce initial gold labels through a model ensemble. *Writing:* we used them for copy-editing, readability, and LaTeX and table formatting. We reviewed and verified all AI-assisted work: generated data and reviewed gold-label subsets passed human review (Section 5), and every formalized entailment query is decided by the deterministic Lean 4 kernel; method outputs are evaluated against human annotations. We take responsibility for the final content, including all text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies grounding verification for generated regulatory artifacts. The evaluation uses a proof-of-concept regulatory-compliance dataset and a public contract-summarization dataset. No personal or demographic data is used, and the method does not intentionally exploit dataset biases; because its translation step is neural, it can still inherit source-data and model biases, which the deterministic kernel boundary reduces but does not eliminate. Grounding establishes consistency with upstream evidence, not the external correctness of that evidence, so human oversight remains necessary before any downstream decision.

REPRODUCIBILITY STATEMENT

The problem formulation is given in Section 3 and the full verification procedure in Section 4; Section 5 specifies the datasets, gold-label protocol, baseline, shared model, and metric definitions. Appendix A documents the LLM prompts used by VERLEAN, and Appendix C gives the synthetic-benchmark construction and error taxonomy. The internal compliance corpus serves as a proof-of-concept testbed and, being proprietary, cannot be released; the public and synthetic testbeds are fully reproducible: upon publication we will release the instantiated 5,550-claim synthetic benchmark, including source identifiers, clean and injected claims, edited spans, error types, and gold labels, together with public-data annotations, full prompts, generated Lean artifacts, and evaluation scripts. These artifacts and the cited public sources support auditing and recomputing the reported public-data results.

REFERENCES

Daneshvar Amrollahi, Jerry Lopez, and Clark Barrett. Faithful autoformalization via roundtrip verification and repair. *arXiv preprint arXiv:2604.25031*, 2026. doi: 10.48550/arXiv.2604.25031. URL https://arxiv.org/abs/2604.25031.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023. doi: 10.48550/arXiv.2302.12433. URL https://arxiv.org/abs/2302.12433.

Anna Bavaresco, Raffaella Bernardi, Leonardo Bertolazzi, Desmond Elliott, Raquel Fernández, Albert Gatt, Esam Ghaleb, Mario Giulianelli, Michael Hanna, Alexander Koller, Andre Martins, Philipp Mondorf, Vera Neplenbroek, Sandro Pezzelle, Barbara Plank, David Schlangen, Alessandro Suglia, Aditya K. Surikuchi, Ece Takmaz, and Alberto Testoni. LLMs instead of human judges? a large scale empirical study across 20 NLP evaluation tasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 238–255. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-short.20. URL https://aclanthology.org/2025.acl-short.20/.

Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. Humans or LLMs as the judge? a study on judgement bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8301–8327. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.474. URL https://aclanthology.org/2024.emnlp-main.474/.

Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. CUAD: An expert-annotated NLP dataset for legal contract review. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021. URL https://arxiv.org/abs/2103.06268.

Elias Horner, Cristinel Mateis, Guido Governatori, and Agata Ciabattoni. Toward robust legal text formalization into defeasible deontic logic using LLMs. *arXiv preprint arXiv:2506.08899*, 2025. doi: 10.48550/arXiv.2506.08899. URL https://arxiv.org/abs/2506.08899.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024. doi: 10.1145/3695988. URL https://doi.org/10.1145/3695988.

Dongwei Jiang, Marcio Fonseca, and Shay B. Cohen. LeanReasoner: Boosting complex logical reasoning with lean. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 7497–7510. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.416. URL https://aclanthology.org/2024.naacl-long.416/.

Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. MARE: Multi-agents collaboration framework for requirements engineering. *arXiv preprint arXiv:2405.03256*, 2024. doi: 10.48550/arXiv.2405.03256. URL https://arxiv.org/abs/2405.03256.

Alan Khoja, Martin Kölbl, Stefan Leue, and Rüdiger Wilhelmi. Automated consistency analysis for legal contracts. In *Model Checking Software*, volume 13255 of *Lecture Notes in Computer Science*. Springer, 2022. doi: 10.1007/978-3-031-15077-7_1. URL https://doi.org/10.1007/978-3-031-15077-7_1.

Kiro. Requirements analysis: Catching requirement bugs before they become code. Kiro Applied Science Blog, 2026. URL https://kiro.dev/blog/deep-spec-analysis/. Accessed: 2026-09-17.

Madhava Krishna, Bhagesh Gaur, Arsh Verma, and Pankaj Jalote. Using LLMs in software requirements specifications: An empirical evaluation. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pp. 475–483. IEEE, 2024. doi: 10.1109/RE59067.2024.00056. URL https://doi.org/10.1109/RE59067.2024.00056.

Wojciech Kryściński, Bryan McCann, Caiming Xiong, and Richard Socher. Evaluating the factual consistency of abstractive text summarization. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pp. 9332–9346, 2020.

Philippe Laban, Tobias Schnabel, Paul N Bennett, and Marti A Hearst. Summac: Re-visiting nli-based models for inconsistency detection in summarization. *Transactions of the Association for Computational Linguistics*, 10:163–177, 2022.

Shuvendu K. Lahiri. Evaluating LLM-driven user-intent formalization for verification-aware languages. In *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design*, 2024. doi: 10.34727/2024/isbn.978-3-85448-065-5_19. URL https://arxiv.org/abs/2406.09757.

Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions. *arXiv preprint arXiv:2507.20439*, 2025. doi: 10.48550/arXiv.2507.20439. URL https://arxiv.org/abs/2507.20439.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=hUb2At2DsQ.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of EMNLP*, 2023.

Laura Manor and Junyi Jessy Li. Plain English summarization of contracts. In *Proceedings of the Natural Legal Language Processing Workshop 2019*, pp. 1–11, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W19-2201.

Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021. doi: 10.1145/3473582. URL https://doi.org/10.1145/3473582.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *Proceedings of EMNLP*, 2023.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Johannes J. Norheim, Eric Rebentisch, Dekai Xiao, Lorenz Draeger, Alain Kerbrat, and Olivier L. de Weck. Challenges in applying large language models to requirements engineering tasks. *Design Science*, 10:e16, 2024. doi: 10.1017/dsj.2024.8. URL https://doi.org/10.1017/dsj.2024.8.

Theo X. Olausson, Alex Gu, Ben Lipkin, Cedegao E. Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger 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*, pp. 5153–5176, 2023.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, 2023.

Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with Symboleo. *Software and Systems Modeling*, 21(6):2395–2427, 2022. doi: 10.1007/s10270-022-01053-6. URL https://doi.org/10.1007/s10270-022-01053-6.

Chenyang Yang, Yike Shi, Qianou Ma, Michael Xieyang Liu, Christian Kästner, and Tongshuang Wu. What prompts don't say: Understanding and managing underspecification in LLM prompts. *arXiv preprint arXiv:2505.13360*, 2025. doi: 10.48550/arXiv.2505.13360. URL https://arxiv.org/abs/2505.13360.

Zhe Ye, Aidan Z. H. Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned formal specification synthesis via traceable refinement. *arXiv preprint arXiv:2604.10392*, 2026. doi: 10.48550/arXiv.2604.10392. URL https://arxiv.org/abs/2604.10392.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Zheng Yuan, Dahua Lin, and Kai Chen. Lean Workbook: A large-scale lean problem set formalized from natural language math problems. *arXiv preprint arXiv:2406.03847*, 2024. doi: 10.48550/arXiv.2406.03847. URL https://arxiv.org/abs/2406.03847.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46595–46623, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Jin Peng Zhou, Charles Staats, Wenda Li, Christian Szegedy, Kilian Q. Weinberger, and Yuhuai Wu. Don't trust: Verify—grounding LLM quantitative reasoning with autoformalization. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=V5tdi14ple.

Taohong Zhu, Lucas C. Cordeiro, and Youcheng Sun. ReqInOne: A large language model-based agent for software requirements specification generation. *arXiv preprint arXiv:2508.09648*, 2025. doi: 10.48550/arXiv.2508.09648. URL https://arxiv.org/abs/2508.09648.

APPENDIX A PROMPTS

This appendix documents the LLM prompts used by VERLEAN (A.1–A.6) and by the two comparison systems it is measured against (A.7–A.8). VERLEAN's own prompts are applied uniformly across the compliance, public, and synthetic evaluations; only the input facts and claims differ. Within VERLEAN, no prompt asks the LLM for an overall VALID/NOT_VALID verdict: prompts request formalizations (a `World`, policy functions, or condition–conclusion queries) or a constrained rewrite, and the Lean 4 kernel decides every formalized entailment query; gap refinement may conservatively leave an assertion unsupported. The baseline judge (A.7) and the *Formalize-then-judge* ablation (A.8) are *not* part of VERLEAN: by design they hand the verdict to the model, which is exactly the trust boundary VERLEAN removes. The system prompts below are lightly abridged for length (repeated worked examples are elided at `[...]`), but the role, task, output schema, and the semantic-strength discipline are reproduced faithfully; full unabridged prompts will be released with the artifacts. Each stage is invoked with a matching user message that supplies the concrete facts or claim text.

A.1 SEMANTIC-MODEL CONSTRUCTION, STEP 1: WORLD-STRUCTURE EXTRACTION

Given the set of cited facts, the LLM extracts only the condition dimensions (enumerated and Boolean fields) that determine which rules apply. It emits no functions at this step.

```
You are a formal methods analyst. Given a set of regulatory facts, extract ONLY
the WORLD STRUCTURE -- the condition dimensions that determine which rules apply.
Do NOT produce any functions/rules -- only the World definition.

## What to extract
- Enum types: dimensions where facts give DIFFERENT rules for different values
  (e.g. "20% for the standard regime, 15% for the reduced regime" -> TaxRegime enum).
- Counterparty jurisdiction as an ENUM (not a bool) whenever the facts distinguish
  among non-residents (treaty vs non-treaty, preferential regimes).
- Payment purpose/use as an enum whenever a fact conditions a rate, exemption, or
  obligation on what the payment is FOR.
- Bool dimensions: yes/no conditions (MUST start with "is_"), only when facts give
  different outcomes for true vs false.

## Material scope dimensions (critical for grounding)
Also include a dimension when a fact states a rule only for a MATERIAL SCENARIO,
even if the facts give no explicit "else" rule. The test: if removing the phrase
would make the rule apply to MORE situations than the fact covered, it is a
material scope and belongs in the World. [...]

## CRITICAL: Keep World SMALL
Only dimensions where the facts EXPLICITLY give different outcomes OR material
scopes needed to prevent over-broad grounding. Aim for the MINIMUM set of
dimensions that captures ALL conditional branching in the facts.

## Output format (strict JSON):
{
  "enums": [{"name": "TaxRegime", "values": ["standard", "reduced"]}],
  "bools": [{"name": "is_b2b", "description": "transaction is B2B (vs B2C)"}]
}
```

A.2 SEMANTIC-MODEL CONSTRUCTION, STEP 2: POLICY-FUNCTION EXTRACTION

For each fact and the fixed `World`, the LLM writes Lean functions formalizing every assertion in that fact. This is the longest prompt because it carries the discipline that keeps a function *exactly as wide as its source sentence*: preserve modality and quantifiers, never invent an `else` branch the fact does not state, and never fold two obligations into one atom. The Boolean convention below is a closed-world simplification that can conflate absence of an obligation with absence of evidence, as discussed in Section 7.

```
You are a formal methods analyst. Given ONE regulatory fact and a FIXED World
structure, produce POLICY FUNCTIONS that formalize EVERY assertion in this fact.

## Your job
Write Lean 4 functions: `def <name> (w : World) : <type> := <body>`
- Return types: Int (rates/amounts, percent x100), Bool (obligations),
  String (classifications). Use if/match on World fields for conditional rules.
- You may ONLY reference `w.<field>` for fields listed in the FIXED World.
  NEVER invent new fields. Quote all String values; use short names, not sentences.
```

```
## COMPLETENESS IS CRITICAL
Read the fact sentence by sentence -- each sentence that states something verifiable
(a rule, value, obligation, prohibition, requirement, qualifier) MUST become a
function. Do NOT skip assertions because they seem minor. [...]

## MODALITY AND QUANTIFIER EXACTNESS -- DO NOT HARDEN SOFT FACTS
Preserve the legal/modal strength and quantifier of the fact. A softer fact is NOT
support for a harder claim. Never formalize as mandatory/all/always:
- "may", "can", "permitted" is not "must" or "required"
- "should", "recommended" is not a legal obligation unless the fact says it is
- "normally", "typically", "generally", "predominantly" is not always/all
- "some", "certain", "selected", "specified", "relevant" is not "all"/"every"/"any"
- "A or B" is not A alone, and is not B alone
But do NOT demote a direct regulatory instruction merely because it is written as
an imperative verb instead of the word "must". [...]

## SCOPE COMPLETENESS -- CAPTURE EVERY QUALIFIER THE FACT SENTENCE STATES
Before writing a function, write down the sentence's SCOPE -- the COMPLETE set of
qualifiers (recipient jurisdiction, payee type, payment type, regime, purpose...).
AND every qualifier that maps to a World field into the function's condition. The
function must be EXACTLY AS WIDE AS THE FACT SENTENCE -- no wider, no narrower:
- TOO WIDE (dropped a qualifier) -> an ungrounded claim later verifies as SUPPORTED.
- TOO NARROW (added an unstated qualifier) -> a faithful claim is wrongly rejected.

## THE `else` BRANCH IS AN ASSERTION -- NEVER FABRICATE IT
`if C then X else Y` asserts that in EVERY world where C is false the result is Y.
If the fact is SILENT on not-C, do NOT invent a plausible value; use a sentinel:
Int -> `else -1`; String -> `else ""`. Bool OBLIGATIONS are the safe case
("must do Z when C" implies not-obligated when not-C). [...]

## DISJUNCTIVE / CONJUNCTIVE OBLIGATIONS -- NEVER FOLD INTO ONE ATOMIC NAME
Do NOT encode "must do A OR B" as a single atom like
`issuer_must_notify_or_reuse_reference` -- the "or" carries logical meaning the
kernel cannot see. Declare a CHOICE dimension, write the disjunction as a real OR
over it, and write each alternative as its own contingent function. [...]

## Output format (strict JSON):
{"functions": [{"name": "wht_rate", "return_type": "Int",
  "body": "if w.is_resident && w.service == .professional then 1000 else 0",
  "description": "withholding-tax rate on professional fees (percent x100)"}]}
```

A.3 CLAIM TRANSLATION INTO VERIFICATION QUERIES

Given the fixed `World` and policy functions, the LLM reads a claim unit and extracts every independently verifiable assertion as a (condition, conclusion) query. Assertions with no same-proposition function are reported as *gaps* (using the claim's own words) rather than paired to a merely related function.

```
You are translating a regulatory text block into Lean 4 verification queries.
Extract ALL independently verifiable assertions as (condition, conclusion) PAIRS.
The kernel checks, for each pair: forall w, condition w -> conclusion w.

## ENCODE THE CLAIM'S OWN STATED PREMISES INTO THE CONDITION
The claim often states the SCENARIO under which its conclusion holds -- especially
NEGATIVE premises ("there is no treaty in force"). Encode that premise into the
condition whenever it maps to a World field. Do NOT leave condition = `true` when
the claim limited its scope, or the kernel will test worlds the claim excluded and
wrongly reject it. [...]

## SCOPE INHERITANCE
A claim sentence often states a SCENARIO once, then makes MULTIPLE assertions about
it. EVERY pair from that sentence must carry the FULL scenario in its condition,
not just the qualifier nearest the assertion.

## COORDINATED OBJECTS -- ONE VERB, MULTIPLE OBJECTS -> ONE PAIR PER OBJECT
"recommended to confirm VAT AND income recognition" is TWO assertions. Distribute
the verb over each conjunct; a conjunct with no matching function becomes a GAP --
never silently dropped.

## Pair vs gap: SAME PROPOSITION, DIFFERENT WORDING -> pair; ADJACENT -> gap.
Test: if the claim's assertion were false, would this conclusion necessarily be
false too? If not, it is a different proposition -> report a GAP. A MODALITY /
QUANTIFIER MISMATCH ("all/must" claim vs "predominantly" function) is a gap, not a
pair. A wrong pair is WORSE than a gap: it verifies something the claim never said.

## Output format (strict JSON): each pair has five fields
```

```
{"pairs": [{"scope": "resident recipient; corporate entities",
   "claim_quote": "no source withholding applies",
   "condition": "w.is_resident_recipient && w.taxpayer_type == .corporate",
   "conclusion": "source_withholding_applies w == false",
   "dropped_qualifiers": [], "reasoning": "..."}],
 "gaps": ["a substantive assertion quoted from THE CLAIM that no function covers"]}
```

A.4 COVERAGE AUDIT

After query extraction, a coverage audit re-reads the claim and lists any verifiable assertion covered by neither an extracted pair nor a gap. It guards against the single-pass extractor silently dropping an assertion, and applies the same (scope, predicate, semantic-strength) matching test; lexical overlap alone is not coverage.

```
You are auditing whether the extracted verification items fully cover a regulatory
claim. Read the claim SENTENCE BY SENTENCE and enumerate every DISTINCT verifiable
assertion. For EACH, decide if a pair or gap above already accounts for it.

CRITICAL -- do NOT be fooled by LEXICAL OVERLAP. A pair that merely shares
vocabulary does NOT cover an assertion stating a DIFFERENT proposition. Test: if
the assertion were FALSE, would the existing pair necessarily be false too? If not,
it is NOT covered. Coverage requires the SAME (scope, predicate) AND the same
semantic strength (quantifier, negation polarity, time, connectives, speech-act
type). [...]

List ONLY the assertions genuinely NOT covered, quoting the claim's own words. If
everything is covered, return an empty list. NEVER include meta-commentary about
the model.

## Output format (strict JSON):
{"missing": ["exact claim words for uncovered assertion 1", "..."]}
```

A.5 GAP REFINEMENT AND SOURCE-SCOPED VERIFICATION

Each uncovered assertion is checked against the evidence alone. The LLM does *not* judge whether the assertion is grounded; it looks for a fact sentence that *directly* states it and, if found, builds a self-contained local `World` so the kernel can decide entailment. If no direct supporting sentence exists, the assertion is left unsupported.

```
A claim asserts something the current policy model does not cover. Your job is NOT
to judge whether it is "grounded" -- check the FACT TEXT and, only if the fact
LITERALLY states it, formalize it into a small self-contained world for the kernel.

## Instructions
1. Find a fact sentence that DIRECTLY states the assertion. DIRECTLY = the fact
   literally says it -- NOT "is consistent with", NOT "implies as a reasonable
   consequence". Match on ALL axes: scope/domain, strength/quantifier, negation
   polarity, time, connectives, speech-act type. If any axis differs materially,
   return supported=false. [...]
   - SCOPE MUST MATCH EXACTLY: do not transplant a fact's attribute onto a
     DIFFERENT entity/type the claim names. Distinguish SERVICE TYPE (what is sold)
     from CUSTOMER/BUYER TYPE (who buys) -- they are different axes.
2. If a direct sentence exists, build a SMALL self-contained world (<= ~6 dims):
   `world` (the dimensions), `fact_functions` (formalize EXACTLY the fact sentence),
   and `condition`/`conclusion` (formalize the CLAIM). Do NOT bake the claim's
   desired answer into a fact_function -- keep fact and claim separate so the kernel
   does real work.
3. If NO fact sentence directly states it -> supported=false.

## Output format (strict JSON):
{"supported": true, "cf_id": "c3", "quote": "exact supporting sentence",
 "world": {"enums": [...], "bools": [...]},
 "fact_functions": [{"name": "...", "return_type": "Int", "body": "..."}],
 "condition": "...", "conclusion": "..."}
or {"supported": false, "reason": "the fact text does not directly state this"}
```

A.6 COUNTEREXAMPLE-GUIDED REGENERATION

When the kernel rejects a claim, VERLEAN renders the failing world as a natural-language repair signal and gives it, with the evidence and the flagged claim, to this rewriter (Section 5.6). The rewriter

never adds information: it may only *narrow* the claim to a qualifier the evidence states or *delete* it. The two moves are not enforced by constrained decoding; the model reports which move it took, and VERLEAN then re-runs the full pipeline on the revision. A *delete* is grounded by construction, and a *narrow* passes only if the kernel re-derives it, so an unfaithful rewrite is caught by re-verification rather than trusted.

```
You are editing one summary in a regulatory knowledge base. A downstream formal
verifier (a Lean kernel that exhaustively checks every scenario) has proven that ONE
claim in the summary is NOT grounded in the cited facts: the claim holds in a specific
scenario ("counterexample world") that the facts do not support.

Your job is to rewrite the summary so that EVERY claim is fully grounded in the cited
facts, using ONLY these two moves on the flagged claim:
  (1) NARROW it -- add the qualifier/condition the facts actually state, so the claim
      no longer over-reaches; OR
  (2) DELETE it -- remove the assertion entirely if it cannot be grounded by narrowing.

Hard rules:
- Do NOT introduce any fact, number, rate, condition, or citation not present in the
  cited facts below. You may only weaken/qualify/remove -- never add new information.
- Do NOT touch the other claims; leave their wording and citations intact. Only the
  flagged claim may change.
- Preserve the summary's structure and inline citations for unchanged claims.
- Prefer NARROW over DELETE when the facts contain a qualifier that makes the claim
  true. Only DELETE when no qualifier in the facts can ground it.

## Output format (strict JSON, no prose outside it):
{"fix_type": "narrow" | "delete",
 "rewritten_statement": "<full summary, with only the flagged claim changed>",
 "rewritten_claim": "<the flagged claim after your edit, or empty string if deleted>",
 "rationale": "<one sentence: which qualifier you added / why you deleted>"}
```

A.7 COMPARISON BASELINE: DIRECT LLM-AS-A-JUDGE

The baseline judge (Section 5) reads one atomic claim and its cited facts and returns a verdict directly, with no formalization and no kernel. It sees the same atomic claim and evidence as VERLEAN, so the comparison isolates the formal check.

```
You are a meticulous grounding analyst for regulatory fact summaries.

A summary is written EXCLUSIVELY from a set of source facts (its cited facts). Your job
is to judge whether a specific CLAIM from the summary is fully grounded by those facts.

## Your task
Given a CLAIM and the CITED FACTS it was summarized from, determine:
- "valid": the claim is FULLY supported by (entailed by) the cited facts. Every factual
  assertion in the claim can be traced to a specific cited fact.
- "not_valid": the claim contains at least one assertion NOT supported by any cited fact
  -- it contradicts a fact, adds information no fact states, or makes a claim no cited
  fact covers (even if plausible or generally true).

## Grounding standard (STRICT)
- "valid" ONLY if EVERY factual assertion is directly stated or directly entailed by at
  least one cited fact.
- If the claim adds ANY information beyond what the facts state -- a number, a rule, a
  classification, a date, an obligation -- it is "not_valid".
- Do NOT use world knowledge to fill gaps. If a claim is plausible but no cited fact
  states it, it is "not_valid".
- Paraphrase and reorganization are acceptable: meaning must be preserved, not wording.

## Output format (strict JSON, no prose, no markdown fence):
{"verdict": "valid" or "not_valid", "reasoning": "brief explanation"}
```

A.8 COMPARISON ABLATION: FORMALIZE-THEN-JUDGE

The *Formalize-then-judge* ablation (Section 5.8) runs VERLEAN's full formalization pipeline, then replaces the kernel with the LLM: it reads the generated Lean and predicts each query's Boolean value instead of evaluating it. It isolates the kernel from the formalization.

```
You are the evaluator in a formalize-then-judge baseline. You receive comment-free Lean 4
code defining finite types, a finite World, functions, and marked #eval queries. Determine
the Boolean value that each marked #eval would return.
```

```
Use only the formal code. Do not use external knowledge or infer any hidden
natural-language claim. For a finite universal implication, account for every represented
World valuation.

## Output format (JSON only):
{"results":[{"query_id":"q0000","value":true},{"query_id":"q0001","value":false}]}
Include every requested query exactly once. Values must be JSON booleans.
```

APPENDIX B EVALUATION DATASET PROFILES

B.1 COMPLIANCE CORPUS

The compliance corpus covers tax, invoicing, payments, and related operational requirements for regulatory-compliance workflows across multiple jurisdictions. Pooled across the selected jurisdictions, the evaluated subset comprises 1,624 atomic claims with complete evidence provenance: 686 fact-to-summary claims and 938 summary-to-requirement claims (Table 1).

The regulatory facts were extracted from external regulatory and legal materials and reviewed by domain experts before consolidation. Structured summaries and requirements were generated by Opus 4.8 under an internal workflow. Graph edges preserve provenance from each summary to its source facts and from each requirement to its authorized summaries, system facts, and technical decisions.

We profile the corpus here in place of releasing it; the public legal-summary and synthetic-contract evaluations remain independently reproducible from their cited datasets.

B.2 PUBLIC LEGAL SUMMARIES

The public transfer evaluation uses the *Plain English Summarization of Contracts* dataset, which pairs source passages with reference summaries. The original collection contains 446 pairs: 85 from TLDRLegal and 361 from ToS;DR. We omit two designated non-evaluation items, leaving 444 units. Each complete summary is evaluated intact against its source passage, with no pre-verification splitting. Median lengths are 59 tokens for source passages and 13 for complete summaries under whitespace tokenization.

B.3 SYNTHETIC CONTRACTS (CUAD)

The synthetic evaluation starts from 300 public CUAD commercial contracts. Their source excerpts support 3,475 certified grounded claims; minimally editing one claim at a time adds 2,075 claims with a single typed grounding error, yielding 5,550 evaluation units. Claims have a median length of 30 whitespace-delimited tokens. Appendix C details benchmark construction.

APPENDIX C SYNTHETIC BENCHMARK CONSTRUCTION

Three prompts construct the synthetic benchmark of Section 5.7. None asks a model for a grounding verdict on the benchmark itself: one generates a near-extractive summary, one independently certifies each claim as grounded (its output, after the human review of Section 5, is the VALID gold), and one injects a single typed error to create a NOT_VALID claim with a known type. The prompts are lightly abridged for length.

C.1 CLEAN-SUMMARY GENERATION

```
You write STRICTLY GROUNDED summaries of legal contracts.

You will receive the full text of one commercial contract. Produce a summary as
a LIST of atomic claims. Each claim must be ONE atomic, self-contained assertion
that is DIRECTLY supported by the contract text.

Write NEAR-EXTRACTIVELY: reuse the contract's own wording as much as possible.
Each claim should closely track a specific sentence or clause of the source --
lightly condensed, never invented.

HARD REQUIREMENTS (each claim is independently judged against the source):
- Every claim must be entailed by the contract. Do NOT add information,
  obligations, parties, numbers, deadlines, or qualifiers not in the text.
- Preserve conditions and qualifiers exactly. If an obligation applies only
  under a condition or only to a specific party/document type, state it.
- Preserve modality exactly (may/must/shall/should/prohibited); do not
  strengthen or weaken normative force.
- Preserve numbers, durations, thresholds, and named entities exactly.
- One assertion per claim -- do not combine two facts.

Aim for {target_claims} claims covering the most important verifiable provisions.

Return ONLY a JSON object:
{ "summary_claims": ["<atomic claim 1>", "<atomic claim 2>", ...] }
```

C.2 INDEPENDENT GROUNDING CERTIFICATION

```
You are an INDEPENDENT grounding judge.

You will receive the full text of one contract and a numbered list of claims from
a summary of it. For EACH claim, decide whether it is fully GROUNDED in the
contract. A claim is GROUNDED only if every part of it is explicitly stated in, or
directly entailed by, the contract. Judge strictly. Mark a claim NOT grounded if it:
- adds information, an obligation, party, number, date, or qualifier not in the source;
- drops or weakens a condition/qualifier the source attaches to the outcome;
- broadens a specific/conditional statement into a universal one;
- strengthens or weakens modality (may/should vs must/shall/prohibited);
- changes a number, threshold, duration, or named entity;
- contradicts the source.
Do not reward plausibility or topical relevance -- only source support. When
unsure, mark it NOT grounded.

Return ONLY a JSON object:
{ "verdicts": [ {"index": <int>, "grounded": true/false,
   "reason": "<one short sentence>",
   "source_quote": "<verbatim supporting span if grounded, else closest span>"} ] }
```

C.3 TYPED ERROR INJECTION

The injector receives the source, the grounded claims, and one error type with a per-type instruction; it rewrites a single best-suited claim to introduce exactly that error.

```
You are constructing a controlled benchmark of grounding errors.

You will receive: the source contract text, a numbered list of GROUNDED claims
from a faithful summary, and ONE error type with an injection instruction. Choose
the SINGLE claim best suited to host this error type, and rewrite ONLY that claim
so it introduces EXACTLY the specified grounding error and becomes UNSUPPORTED by
the source, while staying fluent and plausible.

RULES:
```

```
- Pick the claim where this error type produces a natural, non-obvious error (for a
  numeric error pick a claim with a number; for a modality error one with a modal).
- Change only what the error type requires; keep the rest of the claim's wording as
  close to the original as possible (a minimal edit).
- The result must be a genuine grounding error of the specified type.
- Do NOT fix or improve the claim. Do NOT introduce a different error type.
- If NO claim can host this error type, set "applicable" to false.

Return ONLY a JSON object:
{ "applicable": true/false, "target_index": <int, or -1>,
  "injected_claim": "<rewritten single claim, or empty>",
  "edit_span_original": "<exact original span changed>",
  "edit_span_injected": "<replacement>",
  "why_unsupported": "<one sentence: why the source no longer entails it>" }
```

Per-type injection instruction. The instruction slotted into the injector for each error type:

- **T1 Added information:** insert one plausible on-topic proposition not stated or entailed anywhere in the source.
- **T2 Dropped condition/qualifier:** take a conditionally-supported claim and delete or generalize the guard, so the outcome reads broader than the source allows.
- **T3 Over-broad quantifier:** turn an existential or conditional statement into a universal one (prepend Every/All, drop a case restriction).
- **T4 Modality shift:** replace one modal token with a strictly stronger one (may $\rightarrow$ must, recommended $\rightarrow$ required, should $\rightarrow$ shall).
- **T5 Numeric value:** perturb one numeric or enumerated value (a rate, duration, threshold, or date).
- **T6 Entity/actor swap:** swap the governed entity, party, or class while keeping the outcome (e.g. Licensor $\leftrightarrow$ Licensee, buyer $\leftrightarrow$ seller).
- **T7 Contradiction:** negate one source outcome under the same subject and condition (permitted $\rightarrow$ prohibited, applies $\rightarrow$ does not apply).

APPENDIX D WORKED EXAMPLES

We illustrate VERLEAN on public commercial-contract language, treating NOT_VALID (an ungrounded claim we want to catch) as the positive class. The Lean is lightly simplified; the reasoning is the real verifier behavior. The first example is a correct catch; the second is a translation-fidelity miss that originates in the LLM *translation* step, not the kernel.

D.1 VERIFICATION HARNESS

Every query is decided by the same fixed, claim-independent harness; the LLM authors only the per-artifact `defs` (the `World` fields, the policy functions, and each claim's `cond`/`concl`). The preamble derives `DecidableEq` and `Repr` on the finite types and defines an enumerator that returns `true` iff no world satisfies the condition yet falsifies the conclusion:

```
namespace WorldGrounding

structure World where            -- one field per condition dimension
  is_after_initial_term : Bool
  deriving DecidableEq, Repr

-- fixed, claim-independent: exhaustive forall over the finite World
def grounded (condition conclusion : World -> Bool) : Bool := Id.run do
  let mut ok := true
  for b0 in [true, false] do     -- one loop per field; product of all
    fields
    let w : World := { is_after_initial_term := b0 }
    if condition w && !conclusion w then ok := false
  return ok

#eval grounded cond concl        -- verdict the harness reads off: false
```

A query's verdict is the closed Boolean `grounded cond concl`, which the harness evaluates with `#eval` and reads as `true`/`false`. Because `World` is finite and every function total, this evaluation terminates and is deterministic; the == comparisons inside `cond`/`concl` rely on the derived `DecidableEq`, and the proposition `grounded cond concl = true` is likewise decidable and independently kernel-checkable by `decide`, so the verdict is a machine-decidable object, not a learned judgment. On a `false` verdict the checker re-enumerates the same finite domain to locate the first world with `condition w && !conclusion w` and renders its field values (the derived `Repr` makes a world displayable) into the natural-language repair signal. Locating that witness is a diagnostic pass outside the trusted decision. The two examples below reuse this harness and show only the per-artifact `defs`.

D.2 CORRECTLY CAUGHT: A DROPPED QUALIFIER

Source. "Either party may terminate this Agreement for convenience *only after the initial three-year term*."

Claim. "Either party may terminate this Agreement for convenience." (*Gold:* NOT_VALID; drops the timing qualifier and asserts a right the source grants only after the initial term.)

```
structure World where
  is_after_initial_term : Bool
  deriving DecidableEq, Repr

def may_terminate_for_convenience (w : World) : Bool :=
  if w.is_after_initial_term then true else false

-- claim drops the timing qualifier: condition is unrestricted
def cond  (w : World) : Bool := true
def concl (w : World) : Bool := may_terminate_for_convenience w == true
```

Verdict: rejected (correct). Counterexample $w = \{\texttt{is_after_initial_term} := \texttt{false}\}$: here $\texttt{cond}\, w = \texttt{true}$ but $\texttt{may_terminate_for_convenience}\, w = \texttt{false}$, so the conclusion fails. The checker finds the world the source does not cover (before the initial term), exactly where the broadened claim over-reaches.

D.3 A TRANSLATION-FIDELITY MISS: "NOT REQUIRED" READ AS "PROHIBITED"

Source. "The Distributor's trademark is **not required** to appear on the product packaging."

Claim. "The Distributor's trademark **must not appear** on the product packaging." (*Gold:* NOT_VALID; "not required" $\neq$ "prohibited", so the source supports only the weaker statement.)

```
structure World where
  is_packaged_product : Bool
  deriving DecidableEq, Repr

-- fact keyed on a single "required" predicate
def trademark_required_on_packaging (w : World) : Bool :=
  if w.is_packaged_product then false else false   -- "not required"

-- claim's "must not appear" mapped onto the SAME "required == false"
    predicate
def cond  (w : World) : Bool := w.is_packaged_product
def concl (w : World) : Bool := trademark_required_on_packaging w ==
    false
```

Verdict: passed, but the claim is ungrounded (a miss). The error is upstream of the kernel: the current Boolean target vocabulary and neural translation flattened two distinct modalities (*not required* versus *prohibited*) onto one predicate, so no world separates them. This is the modality-fidelity limit quantified as T4 in Section 5.7; a target vocabulary that keeps *permitted*, *not required*, *prohibited*, and *mandatory* distinct would let the kernel separate them.