ProveML: Inline Claim Markup for Deterministic Verification of AI-Generated Text

Shane Deconinck
abovebeyond.ai
hello@shanedeconinck.be

August 2026

Abstract

Organizations are deploying LLMs in consequential settings under regulatory pressure, but most generated text carries no machine-checkable link between individual claims and the underlying data. Each generation of models produces more fluent output, but none can reliably signal its own uncertainty.

We present ProveML, a lightweight Markdown extension that places AI-generated claims inside a verifiable boundary. Three inline constructs declare entities, link facts to a data source, and check qualitative judgments against composable, pre-declared thresholds:

```
@[student:42]{Alex} scored %[passRate]{5}% and is ?[r: IS_AT_RISK]{at risk}.
```

Verification is deterministic — string equality and numeric comparison against a flat key-value store, with no model in the loop — so it is effectively free (microseconds per claim), explainable, and it can sit inside a generation loop that feeds each error back to the model; the same verifier can audit text it did not generate. Because a verification rate counts only what is inside markup, the verifier also reports *coverage*: the share of the numbers in a text that sit inside a claim at all.

What it requires, costs and cannot do. ProveML applies where the claims are backed by an addressable structured source: without a fact store there is nothing to resolve against. Marked-up responses ran 49–79% longer in characters than the same text without markup. Verification is exact, so the claim states the canonical value, `391035000000`; how the reader sees it ("$391.0 billion") is a formatting rule the store declares next to the field, applied at render time and never by the model. Derived values with no path in the store cannot be bound, and the threshold construct, while specified and tested, was never produced by a model in our runs.

We evaluate three frontier models of August 2026 — Claude Opus 5, Claude Sonnet 5 and DeepSeek V4 Pro — on two domains, a generated educational benchmark and real SEC EDGAR filings, with three runs each and one correction pass. All three produce the markup on every query, put 90–96% of their numbers inside a claim, and verify 87–100% of those claims on the first pass and 92–100% after one correction. What remains is mostly not a wrong number but a property of the language: in "Amir of 5OL has a pass rate of 53%" the nearest entity is the class, and 69% of the residual errors are pupil facts bound to a class by that rule. We make the remedy part of the specification — a fact may name its own record — and measure it: told that rule, the same models verify 95–100% of their claims on the first pass and every finance claim, and the binding residue disappears wherever the rule is followed. The specification, verifier, reference implementation, benchmarks and all run artifacts are published.

Keywords: AI verification, markup language, deterministic verification, structured data, threshold registry, hallucination detection, inline tagging

1 Introduction

Large Language Models (LLMs) produce fluent, contextually appropriate text that is increasingly difficult to distinguish from human-authored content. This fluency, however, coexists with a fundamental reliability problem: **LLMs hallucinate**. They generate text that is plausible but factually incorrect, including fabricated citations, invented statistics, and misattributed claims.

Hallucination is structural, not incidental: calibrated language models must hallucinate at a rate approaching the fraction of facts appearing exactly once in training (Kalai and Vempala, 2024), and standard training pipelines reward guessing over acknowledging uncertainty (Kalai et al., 2025). Regulation no longer adds urgency; it adds exposure. The EU AI Act's transparency duties have applied since 2 August 2026, with the Commission's Article 50 guidelines adopted two weeks before; the Digital Omnibus deferred only the machine-readable marking duty for systems already on the market, to 2 December 2026 (European Parliament and Council of the European Union, 2024; European Commission, 2026; European Parliament and Council of the European Union, 2026). But the Code of Practice that operationalises these duties concerns marking the *origin* of content and says nothing about its accuracy (European AI Office, 2026): a text can be fully compliant and wrong. And the problem has not gone away with better models: the leading legal research tools hallucinated on 17–33% of queries when tested in 2024, despite vendor claims of near-elimination (Magesh et al., 2025), and a 2026 survey of court filings found more than a thousand containing fabricated citations, a number growing year over year (Liu et al., 2026).

The response this paper takes is not to make the model more truthful but to make its claims *checkable*: every marked claim either matches an addressable record or is flagged, deterministically, before anyone reads it. In short, **ProveML is iXBRL for AI-generated text.** Financial reporting solved a version of this problem two decades ago by embedding machine-readable tags in human-readable filings, so a regulator can audit a number without reading the prose around it (XBRL International, 2013). ProveML applies that idea where the author is a model rather than a person, and adds what that setting needs: entity scoping, so a value is bound to the record it describes, and a threshold registry, so a qualitative judgment is a declared predicate rather than a choice of adjective; Figure 1 shows what that buys the reader.

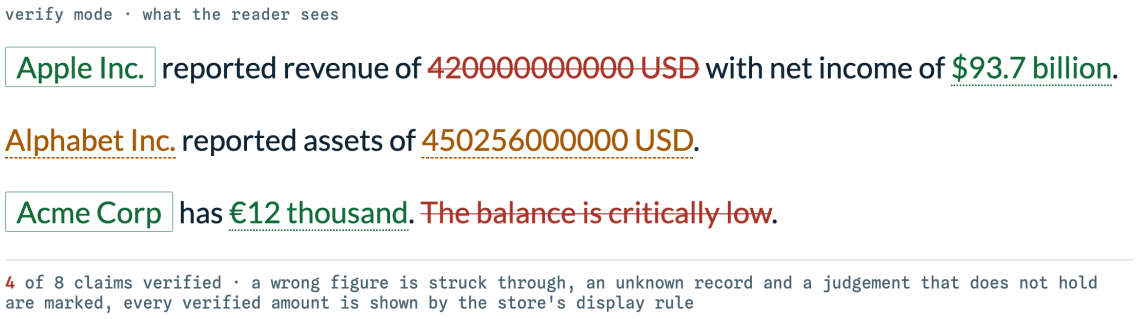

Figure 1: What the reader sees. The same marked-up text after verification: a wrong revenue figure struck through, a claim against an entity the store does not know marked as unverifiable, a judgment whose registered condition does not hold struck through. The verified amounts read "$93.7 billion" and "€12 thousand" because the store declares a display rule for those fields (Section 2); the wrong figure stays as the model wrote it, since only a verified value is formatted. Nothing here required reading the markup, and nothing required a model. Figure 4 shows the companion audit view.

Two limits belong here rather than at the end, because they decide whether the rest is relevant to a given reader. First, exact string equality means the *claim* states the canonical value, `391035000000`, and the model may not round it; the rounded phrasing that financial and journalistic prose normally uses is a rendering the store declares (Section 2), so it never enters the checked text. Second, the threshold construct is specified, tested and demonstrated below, but no model in our runs produced one, so every rate we report measures entity and fact markup only. Section 6 gives the full list, these two included.

This paper is deliberately compact: it gives the idea, the three constructs, what the mechanism costs, and what we measured on three frontier models. The complete specification — verification algorithm, three-valued condition semantics, rendering states, canonicalization rules — is in the accompanying technical report, together with an earlier study (July 2026) of four small local models, a substitution baseline and two ablations, which this paper does not repeat. Both documents are published in the artifact repository alongside every benchmark, run artifact, and the scripts that regenerate every table.[1]

2 ProveML

ProveML extends Markdown with three constructs, using characters (`@`, `%`, `?`) that do not conflict with standard Markdown syntax. LLMs produce Markdown fluently, and existing renderers pass the constructs through unchanged.

```
(* Simple form: entity, then facts follow *)
@[sensor:42]{Sensor X} reads %[value]{29.99} with %[stock]{150} in stock.

(* Scoped form: facts inside entity braces *)
@[sensor:42 "Sensor X"]{reads %[value]{29.99}, %[stock]{150} in stock}

(* Nested scopes: outer entity with multiple inner entities *)
@[facility:7 "Plant North"]{hosts
  @[sensor:42 "Sensor X"]{reading %[value]{29.99}} and
  @[sensor:43 "Sensor Y"]{reading %[value]{18.5}},
  with %[sensorCount]{12} sensors total}
```

Entity references `@[type:id]{name}` declare the subject. The verifier checks the display name against the store (`sensor:42.name`), and the entity becomes the context for the facts that follow. Binding is structural, never controlled by the LLM: a fact inside scoped braces binds to that entity (lexical scope); a fact outside binds to the last simple-form entity at the current depth (linear carry-forward), and closing a scope restores the context that was in force when it opened.

Fact references `%[field]{value}` claim a value. The verifier checks `store`[*entity.field*] = *value* by exact string equality on the store's canonical representation (the store follows below; where the store declares a unit, the claim carries it, as in `%[balance]{-12400 EUR}`). Four outcomes: *verified*, *mismatch*, *unverifiable* (no such field), *no context*. A fact may also name its own record, `%[student:20414.passRate]{53}`, for a sentence whose subject is not the nearest entity; Section 3 shows why that is needed.

[1] https://github.com/abovebeyond-ai/proveml-research — reference implementation: https://github.com/abovebeyond-ai/proveml (npm: `proveml`).

Inference references `?[label: CONDITION]{text}` make a qualitative judgment checkable. The condition names a threshold from a registry defined outside generation; the model cannot invent a comparison value, a bound, or a direction, and an unregistered name is an error rather than a claim. Conditions compose with `AND`, `OR`, `NOT` and can reference earlier labels:

```
?[low: IS_LOW]{critically low score}
?[missing: IS_MISSING]{no recent data}
?[risk: @low AND @missing]{low score with missing data}
```

The fact store is a flat key-value index in the Entity-Attribute-Value pattern; any source with records and fields flattens into it as `type:id.field` → `value`, with optional companion keys for units and nested sub-paths for hierarchical data:

```
account:901.holder                          -> "Acme Corp"
account:901.balance                         -> -12400
account:901.balance._unit                   -> "EUR"
account:901.overdue                         -> 3
account:901.transaction:2026-01.amount      -> 5200
account:901.transaction:2026-01.amount._unit -> "EUR"
```

The store is the single point of trust and the canonicalization point: the guarantee is "consistent with the fact store", not "true", and verification is always relative to an immutable store state (deployments can bind a snapshot identifier to each result). Arithmetic lives in the data layer, not the verifier: a cross-entity difference is materialized as an addressable fact (`region:EU._salesDiff`) and bounded by a registered predicate, so every operand of every comparison is itself auditable. So does presentation: a companion key `revenue._display = currency:USD:1` tells the renderer to show a verified `391035000000 USD` as "$391.0 billion", with the canonical value kept on the element and in the audit view. The claim, the check and the stored text stay exact; only what the eye meets is rounded, by a rule the model never sees (Figure 2). Tolerance in the verifier would have bought the same readability at the price of the guarantee.

Figure 2: One claim, three views. The model writes the canonical value; the store declares the unit and a display rule next to the field; the reader sees the rounded form; the audit view keeps the exact value and says how it was shown. The model never rounds, so it can never round wrong.

The threshold registry draws on clinical reference ranges, JSON Schema validation and monitoring alert rules: domain experts think in named ranges with semantic labels, not operator expressions. Each definition has a name, a field, one predicate from a deliberately small vocabulary (Table 1), an optional unit constraint, a label and a source for audit:

```
IS_LOW:        score lt 25         "critically low"
IS_MODERATE:   score between 25 50 "moderate"
IS_ACTIVE:     status in {A,B,C}  "active"
IS_MISSING:    value is_null       "no data"
MUCH_HIGHER:   _scoreDiff diff_gt 20 "much higher"
IS_NEGATIVE:   balance lt 0 [EUR]  "negative balance"
```

Operator	Display label	Example
Core (numeric comparison)		
`lt`	is less than	`price lt 10`
`lte`	is at most	`rating lte 3`
`gt`	is greater than	`stock gt 0`
`gte`	is at least	`score gte 75`
`eq`	equals	`status eq "active"`
`neq`	is not	`status neq "archived"`
`between`	is between	`score between 25 50`
Extended		
`in`	is one of	`category in {A,B,C}`
`is_null`	is missing	`value is_null`
`diff_gt`	exceeds by more than	`_scoreDiff diff_gt 20`

Table 1: Threshold operator vocabulary. Core operators cover numeric comparison and ranges. Extended operators handle categorical data, missing values, and cross-entity comparison.

Verification resolves each construct against the store — string equality for entities and facts, typed comparison for thresholds — with no model in the loop, so it is exact, explainable and effectively free (Section 4). A judgment the verifier cannot evaluate, because the threshold it names was never registered or the value it needs is missing, is reported as unverifiable, and negating it does not make it verified: `NOT` of an unverifiable condition is still unverifiable. Without that rule a model could get a claim past the verifier by negating a threshold that does not exist. Because errors come back as specific messages ("`%[passRate]{7} in student:42: should be 5`"), verification can sit inside a generation loop that feeds each error back to the model; and because the verifier only reads the text, it can equally audit text it did not generate. A verification rate counts only what is inside markup, so a text that marks one number and leaves nine in prose would score 100%; the verifier therefore also reports *coverage*, the share of standalone numbers that sit inside a claim (years, list markers and code excluded), and in strict mode treats each number outside one as a finding. Figure 3 shows the pipeline; Figure 4 the audit rendering, where each claim carries its proof path. The complete grammar is in Appendix B.

3 What We Measured

Setup. Three models that were frontier at the time of writing (August 2026): Claude Opus 5 and Claude Sonnet 5 (Anthropic, through the Claude Code CLI with default settings), and DeepSeek V4 Pro (`DeepSeek-V4-Pro-0813`, open weights under MIT, served by Together AI). Two benchmarks: 28 English prompts over a generated educational dataset of 741 pupils in

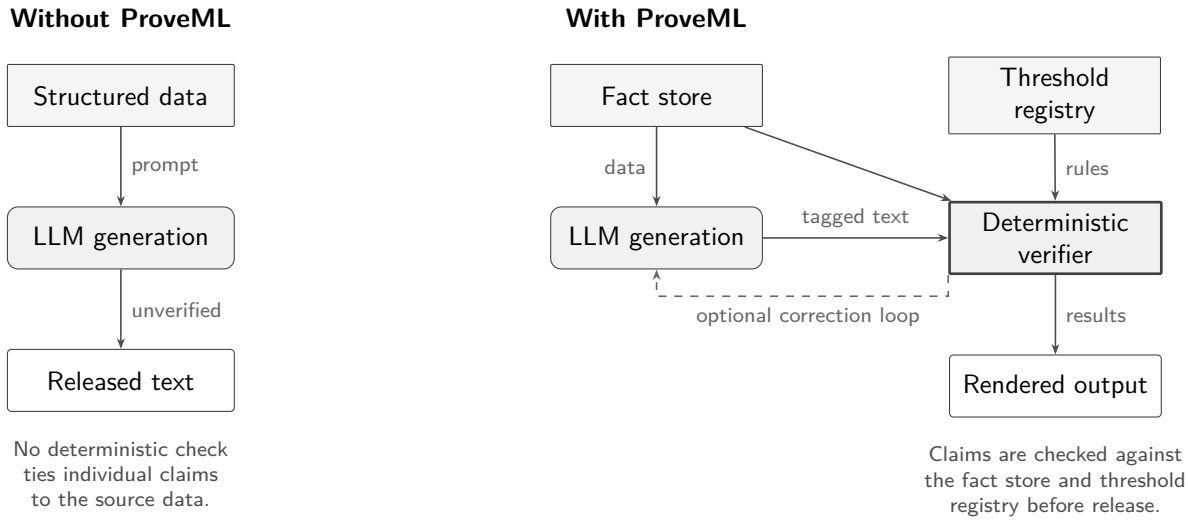

Figure 3: Without ProveML (left): data goes in, text comes out, no verification. With ProveML (right): every claim is checked against the fact store and threshold registry, with an optional correction loop. Rounded boxes denote generative components; squared boxes denote data stores and deterministic processing.

95 class offerings,[2] and 10 English prompts over real SEC EDGAR FY2025 filings (2 entities, 11 fields). Three runs per model and benchmark, one correction pass: the verifier's errors are fed back once, and the corrected answer is accepted only if it verifies better without dropping claims. Verification rate is a per-query macro-average; a response with no markup scores 0%, an empty answer scores 0% rather than being dropped. Context is sliced per prompt from the benchmark's own entity lists — an oracle retriever, which makes the rates an upper bound conditional on retrieval. The system prompt asks for entity and fact markup; no run produced an inference construct, so all rates measure those two. Every number in this section regenerates from the published runs (`experiments/frontier-summary.mjs`, `frontier-residuals.mjs`); the verifier's own conformance test (20 planted errors, all caught) is in Appendix A.

Benchmark	Model	First pass	One correction	Fully verified	Coverage
Education	Claude Opus 5	86.5% ± 0.3	94.8% ± 4.8	23.3/28	91.4%
	Claude Sonnet 5	89.3% ± 2.1	92.2% ± 3.6	21.3/28	92.3%
	DeepSeek V4 Pro	97.2% ± 1.3	99.5% ± 0.8	27.7/28	94.2%
Finance	Claude Opus 5	97.9% ± 0.6	100% ± 0.0	10/10	95.8%
	Claude Sonnet 5	96.5% ± 1.2	96.5% ± 1.2	7/10	89.8%
	DeepSeek V4 Pro	100% ± 0.0	100% ± 0.0	10/10	96.4%

Table 2: Three frontier models, three runs each (mean ± sample sd over runs). First pass and after one correction: per-query verification rate. Fully verified: queries on which every claim verified after at most one correction (mean over runs). Coverage: numbers inside a claim as a share of all standalone numbers in the delivered text, pooled over runs, by the verifier's own definition.

[2]Generated, not de-identified: every record is drawn from a latent-ability model, the school is fictional, and the seeded generator ships with the artifacts. We state this from experience: an earlier draft used a de-identified extract of real records under the label "synthetic"; it was withdrawn, replaced, and every educational result re-measured on the replacement.

Figure 4: Audit mode: the fact-store path checked for each claim is visible inline (`company:aapl.revenue`, `company:msft.debtToEquity`). A reader, or an auditor months later, can take any number back to its record without a model.

Finding 1: the mechanism is within reach of current models. No model, on any of the 342 query-runs, produced an empty answer or an answer without markup, and no call failed. First-pass verification runs from 86.5% to 100%, coverage from 89.8% to 96.4%, and one correction pass lifts Opus 5 by eight points on education and closes finance entirely (Table 2). The open-weight model is the strongest of the three on both benchmarks, at a third of the latency (3.4 s per query against 7.9 and 11.8 s, Section 4). Rate and coverage move together here, unlike in the earlier study of small models, where they diverged by more than thirty points at identical rates (technical report): at the frontier, a model that marks up reliably also marks up nearly everything.

Finding 2: what remains is mostly the binding rule, not the model. Of the 157 errors still standing after the correction pass on the educational benchmark, 108 (69%) have one shape. The sentence names the pupil and then the class — *"@[student:20414]{Amir Janssens} of @[offering:10056]{5OL} has a pass rate of %[passRate]{53}%"* — and linear carry-forward binds the pupil's facts to the class, the nearest preceding entity. The data is right, the English is right, and the verifier reports `offering:10056.passRate` not found. It touched 17 of the 35 non-converged query-runs, and the correction pass could not repair it because nothing in the error message says which entity the fact should have gone to. The scoped form exists for exactly this, and no model used it. We have therefore made a fact able to name its own record, `%[student:20414.passRate]{53}`, which the grammar's `path_field` production always admitted and the verifier now honours, and then measured it (Finding 3). Wrong values are 40 of the 157 (25%), almost all on aggregation prompts that ask for a rank or a count the store does not hold. Two errors are a bound written as a fact (`%[passRate]{60}` before any entity, for "a 60% threshold"): a number that is not a record, which is what the inference construct is for.

Finding 3: told the rule, the models follow it, and the residue goes. We re-ran the study with two sentences added to the system prompt, the binding rule and the cutoff rule, and nothing else changed (Table 3). On finance all three models verify every claim on the first pass in every run. On education Sonnet 5 rises from 89.3% to 99.4%, DeepSeek from 97.2% to 99.2% and to 100% after one correction, and Opus 5 from 86.5% to 95.4%: two of

Benchmark	Model	First pass	One correction	Fully verified	Coverage
Education	Claude Opus 5	95.4% ± 7.5	97.1% ± 4.7	25.7/28	97.9%
	Claude Sonnet 5	99.4% ± 0.2	99.4% ± 0.2	25.7/28	91.1%
	DeepSeek V4 Pro	99.2% ± 1.5	100% ± 0.0	28/28	95.1%
Finance	Claude Opus 5	100% ± 0.0	100% ± 0.0	10/10	98.2%
	Claude Sonnet 5	100% ± 0.0	100% ± 0.0	10/10	94.4%
	DeepSeek V4 Pro	100% ± 0.0	100% ± 0.0	10/10	97.6%

Table 3: The same study under the second prompt, which adds two rules to the first: a fact may name its own record when the sentence names two subjects, and a cutoff from the question is not a fact. Everything else — models, benchmarks, slices, runs, correction budget — is identical to Table 2.

its three runs verify 100% of claims on the first pass, and in the third it reverted, on three pupil queries, to the very phrasing the rule addresses ("Ali Peeters of 6WEWI has a pass rate of ...") without the explicit record, which accounts for the spread. The 108 binding errors of the first study are 16 in the second, all in that one run. What remains elsewhere is small and mixed: a handful of wrong aggregates, and a model asking for a field the store does not hold (`passRateDiff`, `unevaluatedCount`) rather than inventing a number for it, which is the failure the design prefers. The rules now ship with the implementation as a generated system prompt (`proveml/prompt`), built from the store and the registry so the names the model sees are the names the verifier resolves; each rule in it is tied to a measured gap, and this is how we expect the prompt to evolve.

A first pass of this study measured the harness. The context handed to the models named the fields `students` and `rate` where the store held `studentCount` and `passRate`. Every model reproduced what it was shown, and those two names were the two most frequent "field not found" errors of that pass. We record it because the same mismatch was present in the July 2026 study, whose residual-error analysis it inflated, and because it is a general lesson for deployments: the vocabulary the model sees must be the vocabulary the store resolves, or addressability errors will be measured that no model made.

4 Deployment

The evaluation measures whether models can produce verifiable markup. This section reports what it costs to run and what a deployment has to build, because both are load-bearing for anyone deciding whether to adopt the mechanism, and neither follows from the tables above.

Cost of verification: negligible. Verifying the 2,185 claims in the 252 stored final responses of the educational runs against the 4,826-key fact store takes 3.4 ms on a laptop — 0.013 ms per response, 1.5 μs per claim (`experiments/deployment-numbers.mjs -tag frontier` recomputes every figure in this section). Verification is a hash lookup and a comparison per claim, so it scales with the number of claims, not with the size of the store. Against generation latency (3.4 to 11.8 s per query, summed over the correction pass) this is free, which is what allows verification to sit inside the generation loop rather than after it.

Cost of generation: markup and retries. Two overheads are real. Marked-up responses ran 49% (Opus 5), 66% (Sonnet 5) and 79% (DeepSeek) longer in characters than the same text with the markup stripped; the shortest answers pay the highest relative price, because the

markup is a fixed cost per claim. That is a direct output-token cost. The correction pass adds a second: 77% of query-runs needed none, and the mean is 1.23 generations per query (1.06 for DeepSeek, 1.33 for Opus 5). A deployment that allows one pass and no more captures what we measured; the July study found that further passes rarely change the outcome.

What a deployment has to build. The fact store is the work. Flattening records into `type:id.field` is mechanical once the entities are decided, but deciding them is a modelling exercise: a normalized database with joins does not announce what counts as one addressable entity, and that choice determines which claims can be bound at all. Derived values are the recurring case — a count, a difference, a ratio the report computes on the fly — and the rule is that arithmetic belongs in the data layer, materialized as an addressable fact, rather than in the verifier (Section 2). A threshold registry is optional at the entity-and-fact level and small when present: the example registry that ships with the implementation has twenty-one entries, and neither benchmark needed one (no run produced an inference), so its cost falls entirely on the deployments that use it. The vocabulary the model is shown must be the store's vocabulary (Section 3); that is a one-line rule and an easy one to break.

The retrieval assumption, stated plainly. Our headline rates assume retrieval has already succeeded: context slices come from the benchmark's own entity lists, an oracle retriever. The July study's full-context ablation shows what happens without slicing for small models — zero constructs on every query, with the numbers emitted as plain prose instead (technical report); we did not repeat it at the frontier. Context selection is a precondition of the mechanism, and its quality in a real deployment is the largest variable this paper does not measure. Treat the reported rates as an upper bound conditional on retrieval, and budget for the retrieval work accordingly.

Minimal integration. The reference implementation is an npm package whose verifier has no runtime dependencies (only the optional markdown-it plugin needs a peer):

```
import { verifyProveml } from 'proveml/verify';

const store = { 'student:42.name': 'Alex', 'student:42.passRate': 5 };
const text  = '@[student:42]{Alex} scored %[passRate]{5}%.';

const { verified, total, errors } = verifyProveml(text, store);
// -> 2 / 2, no errors
```

A deployment adds three things to an existing LLM call: the store, a system prompt that asks for the markup, and the verifier on the response. The verify-fix loop and the rendering layer are optional on top.

The store's own provenance. The first limitation in Section 6 — verified means consistent with the store, not true — marks the one link the verifier cannot close, and the implementation gives a deployment a discipline for it rather than leaving it to trust. In the review layer (`proveml/review`), every store value carries its evidence: a quote checked verbatim against an archived snapshot of its source, a declared derivation, or a declared absence, because one cannot quote a source not having something. The machine re-checks those links at build time and refuses to emit the review page when any fails; the one link it cannot check — whether a value is a fair reading of its evidence — is judged by a person on that page, and each judgement is saved under a hash of exactly what it judged, so it expires the moment the evidence changes and review shrinks to the diff. The gate is awaitable in a pipeline (`proveml review -await` exits

nonzero while readings are unjudged or flagged), and how a sign-off is attested is an adapter, so a deployment can bind it to a name, a key signature, or a ledger anchor. This paper applies the layer to itself: its related-work characterisations are ProveML claims against a citation store whose evidence readings ran through exactly this page, quote by quote.

5 Related Work

The idea of tagging human-readable text for machine resolution is old: RDFa binds spans of prose to entities in a structured vocabulary and assumes the author meant it (W3C, 2015); iXBRL embeds machine-readable tags in financial reports for automated audit (XBRL International, 2013). ProveML adds the verdict to that lineage — not what a span refers to, but whether the claim survives comparison with the record. Among recent systems, Proof-Carrying Numbers (Solatorio, 2025) is the nearest neighbor (claim-bound numeric tokens, deterministically verified in the renderer, with declared tolerance policies; its published description does not cover entity scoping or a named vocabulary for qualitative judgments), and SymGen (Torroba Hennigen et al., 2024) takes the opposite strategy: the model emits references and a parser substitutes the values, so a wrong number is impossible and so is reporting one; the technical report compares the two on identical runs. Claim-locked reporting (Fan et al., 2026) is the 2026 form of that strategy — the numbers, their direction and the permitted strength of language are fixed before the model writes connective prose — and shares ProveML's premise that qualitative wording must be tied to a declared threshold, while giving up the ability to audit text it did not produce. VeriFin (Hall et al., 2026) checks financial claims against XBRL facts with an SMT solver, placing the arithmetic in the verifier where ProveML places it in the data layer. Yang et al. (2026) measure the failure class our residual errors belong to — values miscited from a table the model was shown — and detect it with a trained critic; ProveML detects it with a lookup. Fact verification against evidence has a canonical benchmark lineage — FEVER (Thorne et al., 2018), TabFact (Chen et al., 2020), FEVEROUS (Aly et al., 2021) — in which a trained model judges support; ProveML sits outside it by making the judgment a lookup. Probabilistic faithfulness checkers, academic (SummaC, Laban et al., 2022; AlignScore, Zha et al., 2023; MiniCheck, Tang et al., 2024; HallDetect, Oukelmoun et al., 2026) and industrial, score responses after generation; Evergreen (Lee et al., 2026) verifies aggregate claims over relational data as queries, which ProveML can only bind once the aggregate is materialised as a fact; structured inline citation (Yeginbergen et al., 2026) and decode-time grammars whose reference slots admit only declared names (Zhang et al., 2026) are the constrained-generation route to the same end, and the natural next step for ProveML (Section 6). To our knowledge, no existing framework combines inline claim markup, deterministic mismatch detection against structured data, and composable threshold inference in a single system.[3] Table 4 places the neighbours; the technical report discusses each in full.

6 Discussion and Limitations

The deeper value is not the correction loop but the boundary itself. With ProveML markup, every marked claim carries a verification status and an addressable path, so a reader — or an auditor months later — can ask of any number where it came from and get an answer that does not depend on a model. What changes is the unit of accountability: from "was this report right" to "was this claim right, and against which record". Because entity references are structural

[3] We screened the titles and abstracts of the 4,617 papers in the ACL 2026 main, short, findings and industry volumes against concept patterns for claim-level attribution, structured-data faithfulness, symbolic or deterministic verification, provenance and markup schemes, then read the resulting candidates by hand. The sweep was not preserved as a runnable artifact, so this is the result of a search rather than an exhaustive negative.

Approach	Category	Determ.	Structured	Inline	Inference
FActScore, SAFE, FacTool, HallDetect	Fact-checking	–	–	–	–
Guardrail and grounding scorers	Guardrails	–	–	–	–
Schema validators, decode-time grammars	Schema	✓	–	–	–
ClaimDB, Evergreen, VeriFin	Data-grounded	✓	✓	–	–
FullCite	Inline citation	–	–	✓	–
iXBRL*	Inline tagging	✓	✓	✓	–
SymGen	Inline tagging	✓	✓	✓	–
PCN	Inline tagging	✓	✓	✓	–
Claim-locked reporting	Inline tagging	✓	✓	✓	Partial
FinGround	Hybrid post-hoc	Partial	✓	–	–
ProveML	**Inline tagging**	✓	✓	✓	✓

Table 4: Comparison across representative verification approaches. Determ. = deterministic. Structured = against structured data. Inline = claim-level markup in text. Inference = composable threshold checks within natural language text. The guardrails row reflects the predominant probabilistic grounding mode (Bedrock, Azure Groundedness, Vectara HHEM and similar); Amazon Bedrock's separate Automated Reasoning feature adds formal policy checks but not inline claim markup. Claim-locked reporting ties language strength to statistical thresholds but does not compose them. FinGround recomputes arithmetic claims deterministically but decomposes and types them with a model, hence Partial. *XBRL's formula linkbase provides document-level calculation and assertion checks; the iXBRL specification itself does not include claim-scoped inference.

and display text is separate from the identifier the verifier resolves, a deployment can hand the model pseudonymous identifiers and substitute names at display time — a property the syntax makes available, not one we evaluated.

- **Consistency, not truth.** ProveML verifies that claims match the fact store, not that the fact store is correct. If the source data is wrong, verified claims will be wrong. The review layer described under Deployment gives that boundary a discipline — evidence per store value, a human sign-off saved under a hash of exactly what it judged, expiring when the evidence changes — but does not remove it: a fair reading is a judgement, not a proof.
- **The right name, not necessarily the right record.** An entity verifies when the name the reader sees equals the name stored at the id the model chose; nothing checks that the model chose the right id. If two records of one type share a name, a wrong id renders identically to the right one and only the audit path tells them apart. The verifier therefore measures whether a rendered subject is unique in the store (`subjectUnique`: true, false with the other paths, or unknown when the source cannot be enumerated), `doctor` warns on duplicate names, strict mode makes an ambiguous subject a finding, and the render marks it. Where subjects are identifiers by construction, as on a ledger, the gap closes; where they are personal names, it does not, and the marker is what remains.
- **No unit conversion.** Units are supported as nullable metadata with exact string matching. Semantic unit equivalence (e.g., mg/dL vs mmol/L) and standardized vocabularies (UCUM, ISO 4217) are deferred to future work.
- **Evaluation scope.** The evaluation covers two domains (generated education, real SEC finance) across 38 prompts, 3 models, and 3 repeats, with an oracle retriever and one correction pass. Larger schemas, more domains, and retrieval under realistic conditions are not covered. The explicit-record fact form was introduced after the first study and measured in the second (Finding 3); whether a model follows the rule on every run is not guaranteed, as one Opus 5 run shows.
- **Inference constructs are unmeasured.** The system prompt used throughout the evaluation asks for entity and fact markup only. No run produced an inference construct, so

every rate we report is a rate over entities and facts, and the threshold registry — the part of the design that constrains qualitative language — is exercised by the specification, the test suite and the worked examples, but not by the study. How reliably models emit `?[label: THRESHOLD]{text}`, and how often they reach for a threshold that fits the sentence rather than the data, is an open question this paper does not answer.

- **Canonical claims.** The model may not round: a claim says `391035000000`, never "$391 billion". The reader can still see the rounded form, because the store may declare a display rule per field (`_display`) that the renderer applies to verified values; but a value the model itself rounded is a mismatch, and prose that quotes the rounded figure outside a claim is unverified prose. The rule is on the store's side of the boundary on purpose: a tolerance in the verifier would make "about 18" verify against 18.5 and turn the guarantee into an approximation.
- **Prose outside markup.** ProveML verifies what is inside markup constructs. Text outside the markup is not checked. The verifier now measures the numeric part of this gap as coverage (90–96% in Section 3) and can refuse a number outside any claim, but a qualitative sentence with no number in it remains invisible to it. And a set of verified claims does not verify the sentence around them: Chan et al. (2026) argue that formally sound conclusions invite unsupported inferences by the reader, and nothing in ProveML's verdict speaks to what the prose between two verified claims implies.
- **Malformed input.** The tokenizer silently skips constructs with unmatched brackets or braces. Robustness to diverse malformed markup deserves further stress-testing.
- **Adversarial generation.** ProveML is designed against models that make mistakes, not against a model trying to mislead. A generator that wanted to could choose a threshold that holds and wrap misleading words around it, mark only the claims that verify and leave the damaging ones as prose, or select a true fact that misrepresents the record it came from. Each of these survives verification because the verifier reads paths and values, not intent. We neither measure nor defend against this.
- **Group claims.** Inferences evaluate against a single entity context. A claim like "5TW, 4B, and 6WE all have low coverage" cannot be verified as one inference; it must be decomposed into one inference per entity.

Future work follows from the limits: derived-value claims, standardized unit vocabularies, an abstention metric for unsupported queries, and constraining decoding to well-formed ProveML with only existing paths — grammar-constrained generation (PICARD, Scholak et al., 2021; Synchromesh, Poesia et al., 2022) makes that an application of known technique rather than a research program, and it would remove much of what we measure as addressability error.

7 Conclusion

ProveML places AI-generated claims inside a verifiable boundary. Three inline constructs link every claim to an addressable path in a data source. Verification is deterministic, with no model in the loop. An optional threshold registry extends this to qualitative judgments through composable primitives.

The evaluation (Section 3) says that the mechanism is within reach of current models: three frontier models produce the markup on every query, cover 90–96% of their numbers with claims, and verify 92–100% of those claims after one correction. What remained was a property of the language more than of the models; the specification changed in response, and with the change in the prompt the residue went where the rule was followed. The earlier study of small local models (technical report) says the complement: below the frontier, whether a model produces verifiable markup at all depends on the shape of the request more than on the size of the model.

What LLMs provide is open-ended generation. What ProveML adds is a controllable, auditable

boundary around that generation. For humans, the intended benefit is visual triage: verified claims carry a status, and attention shifts to what is unverified. We have not measured whether this reduces human verification effort, and the evidence from adjacent systems is mixed: Torroba Hennigen et al. (2024) report their user study reduced average verification time by 20%, while Mehrotra et al. (2026) report a 21-participant study in which verification and correction effort did not differ significantly from their baseline, even though their system reduced hallucination. Which of the two ProveML's rendering resembles in practice is an open empirical question. For agent loops the benefit is more direct and does not depend on human factors: specific error feedback ("`%[glucose]{143 mg/dL} in patient:308: should be 142 mg/dL`") enables targeted self-correction, which is what the verify-fix loop measures in Section 3. The combination is what neither open-ended generation nor static verification offers alone: open-endedness with determinism.

Declaration of Generative AI Use

Generative AI tools (Claude, GPT) were used during manuscript preparation for drafting, revision, literature search, and code review. All AI-generated content was reviewed, verified, and edited by the author. No generative AI was used to create or alter images or artwork. The author takes full responsibility for the content of the published article.

References

R. Aly, Z. Guo, M. Schlichtkrull, J. Thorne, A. Vlachos, C. Christodoulopoulos, O. Cocarascu, and A. Mittal. FEVEROUS: Fact extraction and VERification over unstructured and structured information. In *Proc. NeurIPS Datasets and Benchmarks Track*, 2021. arXiv:2106.05707.

Jason Chan, Robert Gaizauskas, and Zhixue Zhao. Position: Logical soundness is not a reliable criterion for neurosymbolic fact-checking with LLMs. arXiv:2604.04177, 2026.

W. Chen, H. Wang, J. Chen, Y. Zhang, H. Wang, S. Li, X. Zhou, and W. Y. Wang. TabFact: A large-scale dataset for table-based fact verification. In *Proc. ICLR*, 2020. arXiv:1909.02164.

European AI Office. Code of practice on transparency of AI-generated content. June 2026, https://digital-strategy.ec.europa.eu/en/faqs/signing-code-practice-transparency-ai-generated-content, 2026.

European Commission. Guidelines on the implementation of the transparency obligations for certain AI systems under Article 50 of the AI Act. Adopted 20 July 2026, https://digital-strategy.ec.europa.eu/en/library/guidelines-transparency-obligations-providers-and-deployers-ai-systems, 2026.

European Parliament and Council of the European Union. Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (artificial intelligence act). Official Journal of the European Union, 2024.

European Parliament and Council of the European Union. Regulation (EU) 2026/1744 of 8 july 2026 amending regulations (EU) 2024/1689, (EU) 2018/1139 and (EU) 2023/1230 as regards the simplification of the implementation of harmonised rules on artificial intelligence (Digital Omnibus on AI). Official Journal of the European Union, https://eur-lex.europa.eu/eli/reg/2026/1744/oj/eng, 2026.

Xiao Fan, Jingyuan Li, Hongbin Guo, Yubo Han, and Yi Zhang. Provenance before prose: Claim-locked reporting. arXiv:2608.25336, 2026.

Bethel Hall, Sachi Shome, and William Eiers. VeriFin: A neurosymbolic framework for verifying LLM-generated financial claims. arXiv:2608.10213, 2026.

A. T. Kalai and S. S. Vempala. Calibrated language models must hallucinate. In *Proc. STOC*, 2024. arXiv:2311.14648.

A. T. Kalai, O. Nachum, S. S. Vempala, and E. Zhang. Why language models hallucinate. OpenAI, arXiv:2509.04664, 2025.

P. Laban, T. Schnabel, P. N. Bennett, and M. A. Hearst. SummaC: Re-visiting NLI-based models for inconsistency detection in summarization. *Transactions of the Association for Computational Linguistics*, 10:163–177, 2022. doi: 10.1162/tacl_a_00453.

Jiwon Lee, Seokhyun Han, Rathijit Sen, Jinsoo Yeom, Ugur Cetintemel, and Anindya Datta. Evergreen: Efficient claim verification for semantic aggregates. arXiv:2604.26180, 2026.

Patty Liu, Dominik Stammbach, and Peter Henderson. Who checks the citations? benchmarking legal hallucination detection. arXiv:2606.21155, 2026.

V. Magesh, F. Surani, M. Dahl, M. Suzgun, C. D. Manning, and D. E. Ho. Hallucination-free? assessing the reliability of leading AI legal research tools. *Journal of Empirical Legal Studies*, 22(2):216–242, 2025. doi: 10.1111/jels.12413.

N. Mehrotra, A. Kumar, S. Gulwani, A. Radhakrishna, and A. Tiwari. TEN: Table explicitization, neurosymbolically. In *Proc. ACL Industry Track*, 2026. doi: 10.18653/v1/2026.acl-industry.138.

Ismail Oukelmoun, Nasredine Semmar, and Gaël De Chalendar. HallDetect: Decomposed entailment for factuality checking. arXiv:2608.05823, 2026.

G. Poesia, O. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *Proc. ICLR*, 2022. arXiv:2201.11227.

T. Scholak, N. Schucher, and D. Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proc. EMNLP*, pages 9895–9901, 2021. doi: 10.18653/v1/2021.emnlp-main.779.

Aivin V. Solatorio. Proof-carrying numbers (PCN): A protocol for trustworthy numeric answers from LLMs via claim verification. arXiv:2509.06902, 2025.

L. Tang, P. Laban, and G. Durrett. MiniCheck: Efficient fact-checking of LLMs on grounding documents. In *Proc. EMNLP*, 2024. doi: 10.18653/v1/2024.emnlp-main.499.

J. Thorne, A. Vlachos, C. Christodoulopoulos, and A. Mittal. FEVER: a large-scale dataset for fact extraction and VERification. In *Proc. NAACL-HLT*, pages 809–819, 2018. doi: 10.18653/v1/N18-1074.

L. Torroba Hennigen, S. Z. Shen, A. Nrusimha, B. Gapp, D. Sontag, and Y. Kim. Towards verifiable text generation with symbolic references. In *Proc. COLM*, 2024. arXiv:2311.09188.

W3C. RDFa 1.1 primer — third edition. W3C Working Group Note, https://www.w3.org/TR/rdfa-primer/, 2015.

XBRL International. Inline XBRL part 1: Specification 1.1. https://www.xbrl.org/specification/inlinexbrl-part1/rec-2013-11-18/inlinexbrl-part1-rec-2013-11-18.html, 2013.

Yuqing Yang, Qi Zhu, Zhen Han, Boran Han, Zhengyuan Shen, Shuai Wang, Vassilis N. Ioannidis, and Huzefa Rangwala. When LLMs read tables carelessly: Measuring and reducing data referencing errors. arXiv:2606.32029, 2026.

Anar Yeginbergen, Amelie Wührl, Anna Rogers, and Rodrigo Agerri. Explicit evidence grounding via structured inline citation generation. arXiv:2606.07130, 2026.

Y. Zha, Y. Yang, R. Li, and Z. Hu. AlignScore: Evaluating factual consistency with a unified alignment function. In *Proc. ACL*, pages 11328–11348, 2023. doi: 10.18653/v1/2023.acl-long.634.

Shuoming Zhang, Ruiyuan Xu, Haofeng Li, Qiuchu Yu, Yangyu Zhang, Chunwei Xia, Xiaobing Feng, Chenxi Wang, Huimin Cui, and Jiacheng Zhao. Decode-time grammars: Constrained LLM generation over a refinement order of grammar fragments. arXiv:2607.18357, 2026.

A Error Detection Rate

Setup. The critical question for ProveML is not whether the LLM generates correct output, but whether ProveML *detects* incorrect output. To test this, we injected 20 deliberate errors into otherwise valid ProveML text and measured whether the verifier caught each one.

Results. The 20 errors span six categories: wrong values (5), wrong entity (3), no context (2), wrong threshold (4), cross-entity swaps (2), and subtle errors including trailing spaces and swapped entity order (4). ProveML detected all 20, yielding a **100% error detection rate**. This is a consequence of deterministic equality checking: if the claimed value does not exactly match the fact store, the claim is flagged regardless of plausibility. The experiment validates that the implementation behaves as specified — it is a conformance test, not a detector benchmark — and it is one-sided: it measures detection of planted errors, not false positives on correct-but-reformatted values (a canonicalization difference such as `18.50` vs. `18.5` is flagged despite numeric equality; see Section 6). Robustness to malformed LLM markup is partially addressed by the correction-loop results in Section 3 but deserves further stress-testing.

B ProveML Grammar (EBNF)

The following EBNF captures the ProveML constructs as implemented in the reference parser. It is intended as a starting point for formal treatment, not a normative specification; the authoritative definition is the reference implementation and test suite. Where the two diverge, the reference tokenizer is deliberately *more permissive* than this grammar — for example, it accepts identifiers beyond the character classes below and tolerates a missing space after the inference label colon — so the grammar describes the canonical surface form generators should produce, while the tokenizer accepts sloppier variants of it. One exception runs the other way: the tokenizer brace-counts entity display text, so an unbalanced open brace (admitted by the `display_text` production) causes the construct to be skipped.

```
(* ProveML constructs inside Markdown inline content *)
entity_ref_simple = "@[" , entity_path , "]{" , display_text , "}" ;
entity_ref_scoped = "@[" , entity_path , " " , DQUOTE ,
                    entity_name , DQUOTE , "]{" ,
                    scoped_content , "}" ;
entity_ref        = entity_ref_simple | entity_ref_scoped ;

fact_ref          = "%[" , field_name , "]{" , value_text , "}" ;
inference_ref     = "?[" , label , ": " , condition , "]{" ,
                    display_text , "}" ;

(* Paths and identifiers *)
entity_path       = identifier , ":" , entity_id ;
entity_id         = ( letter | digit ) ,
                    { letter | digit | "_" | "-" } ;

field_name        = field_segment , { "." , field_segment } ;
field_segment     = meta_field | path_field | identifier ;
meta_field        = "_" , identifier ;
path_field        = identifier , ":" , entity_id ;

label             = identifier ;
identifier        = letter , { letter | digit | "_" } ;

(* Text content *)
entity_name       = { char - '"' - ']' } ;
display_text      = { char - "}" } ;
scoped_content    = { char } ;   (* parsed recursively *)
value_text        = { char - "}" } ;

(* Condition grammar *)
condition         = or_expr ;
or_expr           = and_expr , { " OR " , and_expr } ;
and_expr          = unary_expr , { " AND " , unary_expr } ;
unary_expr        = [ "NOT " ] , primary ;
primary           = threshold_name
                  | threshold_name , "(" , path , ")"
                  | "@" , label ;

(* Terminals *)
threshold_name    = uppercase , { uppercase | digit | "_" } ;
path              = entity_path , "." , field_name ;
DQUOTE            = '"' ;
letter            = "a"-"z" | "A"-"Z" ;
uppercase         = "A"-"Z" ;
digit             = "0"-"9" ;
```