Gen-AI as a Formal Requirements Engineering Partner: A Case Study in Verified Parser Design

Nicolas Rouquette[1][0000−0003−3137−8690]

Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, CA 91109, USA
nfr@jpl.nasa.gov

Abstract. We report on using generative AI (Gen-AI) as tooling in formal requirements engineering, building a verified YAML 1.2.2 parser in Lean 4 over three architectural iterations. The work decomposes into three RE activities, each exposing a different failure mode: *eliciting* formal requirements from an English prose specification with 211 parameterized BNF rules, *validating* them by attempting capstone proofs (missing requirements no tests reveal), and *auditing* the proofs themselves so that they actually constrain the code (*the fibration gap*). We identify four Gen-AI roles—*specification analyst*, *architecture critic*, *proof engineer*, *pattern analyst*—each realised through a distinct prompting strategy, and a mechanically-checkable meta-test, the Fibration Gap Metric, that distinguishes load-bearing capstones from hollow ones. "Partner" denotes integrated tooling, not autonomy: the choice of which theorems to state remained human-driven.

Keywords: Formal requirements · Generative AI · Verified parsing · Lean 4 · YAML · Safety-critical systems

1 Introduction

YAML is a data-serialization language widely used in safety-critical systems. Its anchor/alias expansion feature creates denial-of-service attack vectors ("billion laughs" [6]) and language-specific tags create arbitrary code execution vulnerabilities. Production parsers carry supply-chain risk because *unknown vulnerabilities may exist in parsers that pass all tests.* The YAML 1.2.2 specification [1] is a ∼65-page English prose document with 211 grammar productions. While the specification is deliberately implementation-agnostic, nearly twenty existing implementations interpret it differently and only a handful pass the official test suite[1]. Turning that prose into a mechanically-checkable specification is a textbook RE problem: an ambiguous reference, no executable oracle, no agreement on what the specification means. Our thesis is that Gen-AI tooling acts as a *force multiplier* on three RE activities—encoding, validation, audit—without

[1] https://matrix.yaml.info/

changing the asymmetry that makes formal RE hard: writing the implementation took weeks; validating that the formalized specification captured the prose's intent took months. This paper reports those activities, the failure modes each exposed, and the artifacts that came out of them. We used Claude Code with VS Code over five months and three iterations [3].

2 Three Architectures, Three Lessons

We distinguish *architectural* bugs (a property is unprovable by construction under the chosen design) from *incidental* bugs (a property is provable but the code happens to break it).

Attempt 1: Monadic Parsec (2 weeks). Our first attempt [3] used `StateT YamlContext Parser` over Lean's Parsec. It passed a few hand-written regression tests and lacked sufficient functionality to handle the official YAML test suite [2]. Parsec's `attempt` backtracks string position but does *not* restore user-defined state (indentation, flow context, anchors)—rediscovered as a bug five times, once regressing 230 passing guards to 7. This is *architectural*, not a misread library contract: even reading the contract correctly, a state-consistency theorem is unprovable by construction under any implementation that uses Parsec backtracking. Testing missed it because some inputs happened to leave state restorable.

Attempt 2: Parser combinators (2 weeks). The `lean4-parser` library [4] offered explicit state threading, but hit a wall with *context-sensitive tab processing* (YAML §6.2): in a scannerless parser, character classification depends on parser state which depends on previously classified characters—a circular dependency surfacing only when proving classification consistency. Gen-AI proposed the resolution: separate scanning from parsing.

Attempt 3: Scanner + token parser (9 weeks). The successful design [3] separates a *scanner* (char $\rightarrow$ token, $\sim$3,000 lines) from a *token parser* (token $\rightarrow$ AST, $\sim$1,350 lines), with a formal specification ($\sim$2,700 lines) as an independent artifact, enabling *layered verification*: character predicates, token-level invariants (`Scannable`), and specification-level predicates (`Grammable`). An *anti-drift* mechanism couples each runtime Bool predicate with a specification Prop via an iff theorem; if either drifts, the build breaks. Table 1 summarises the three attempts.

3 Four Gen-AI Roles in the RE Workflow

The four roles below were realised through distinct prompting strategies, each with a different brief, working set of files, and acceptance criterion. None was self-assigned by the model.

Role 1: Specification analyst. Prompts of the form "translate §X.Y into a Lean predicate; cite the production identifier" produced inductive definitions plus traceability annotations; the human validated each. Subsequent capstone proofs

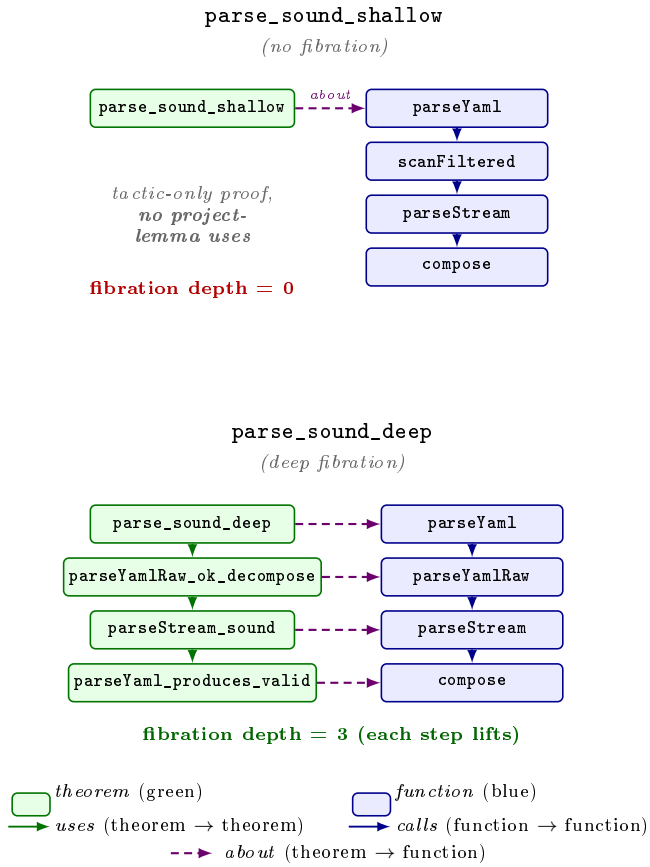


Fig. 1. Worked example of the Fibration Gap Metric on two soundness headlines that both type-check. In each panel the proof DAG (green theorem boxes, left) sits opposite the call DAG (blue function boxes, right); the dashed purple *about* arrow points from a theorem to the function it mentions. *Top:* `parse_sound_shallow` mentions only `parseYaml` in its type and cites no project lemma in its proof; only one *about* edge exists, and the proof DAG has nothing to mirror the call DAG with. *Bottom:* `parse_sound_deep` mentions every pipeline stage in its type and cites a lemma about each in its proof. Each *uses* edge in the proof DAG lifts a *calls* edge in the function DAG via the *about* relation (a commuting square at every step), making the theorem a discrete Grothendieck fibration over the call graph—a *verification canary* that any behavioural change in the implementation must invalidate.

Table 1. Three attempts (4-month total). Compliance is on the 337-test YAML suite [2]; regression tests are hand-written. Attempt 1 was abandoned before test-suite evaluation. The YAML test suite has 406 tests; 48 are skipped since they are about YAML 1.3 (not yet released), leaving 358 relevant tests for YAML 1.2.2 (latest release as of May 1, 2026).

Metric **Date**	**Attempt 1** **Parsec** **2026-02-14**	**Attempt 2** **Combinators** **2026-02-26**	**Attempt 3** **Scan+Parse** **2026-05-01**
yaml-test-suite	None	279	358 (100%)
Regression tests	5/5	1,335/1,337	4,243/4,243
Functions	74	251	539
Proved theorems	9	650	2,821
Theorems w/ Sorries	0	3	7
Architectural failure	state backtrack	tab context	—

caught off-by-one errors that the annotations did not. A *doc-verification-bridge* tool [7] classifies each declaration via Lowe's Four-Category Ontology [8]—*kinds* (inductive types and structures), *attributes* (predicates and propositions), *objects* (data values), *modes* (instantiated theorems)—and infers from each elaborated AST what each theorem assumes, proves, and depends on. In Attempt 3 it revealed 13 of 19 grammar definitions had zero connecting theorems and identified 461 *bridging theorems* (out of 2,934) that connect Bool computations to Prop specifications, feeding proof blueprints [9].

Role 2: Architecture critic. Given all three codebases, Gen-AI classified unfixed bugs as architectural or incidental and proposed a redesign when the architectural count exceeded a threshold—how Attempt 2's tab-context issue was diagnosed as a verification barrier rather than a proof-technique gap.

Role 3: Proof engineer. The human identified candidate properties (from compiler-engineering experience); Gen-AI proposed proof skeletons and iterated on type-checker feedback. This produced 2,934 theorems about 539 definitions [7].

Role 4: Pattern analyst. Asked to scan failed proofs for *recurring shapes*—structural features of code that obstruct *any* proof technique—Gen-AI returned five classes (Lean examples, each generalising): (1) *state updates before lemmatised calls* (the lemma is stated for the original state, the goal mentions the updated one); (2) *reduction-strategy interference* (case analysers that reduce to head-normal form, e.g., `split` via WHNF, can target inner constructs and produce silently wrong sub-goals—introduce a fresh variable for the intended target); (3) *branch-product explosion* (N sequential dispatches yield $O(2^N)$ goals; sub-computation extraction cut a ∼320-line proof to ∼30); (4) *semantic impasse from missing invariants* (a goal $x{+}1 = x$ is not a tactic gap, it is an unrecorded invariant); (5) *predicate polarity mismatch*—e.g., `Scannable input true` encodes "every token, including a closing bracket, has been scanned", yet composing two positively-typed lemmas does not yield the closed form; the closing-bracket wit-

ness must be an explicit hypothesis at the API boundary. Gen-AI proposed mitigations: `generalize` before reduction-driven case analysis, sub-computation extraction, and Wadler-style "free theorems" [5] as refactoring guards.

4 The Fibration Gap Metric: Auditing the Audit

Lean will accept a soundness theorem whose proof never descends into the parser; that theorem survives a refactor that genuinely breaks parsing. When Gen-AI generates both code and proofs, such hollow theorems are easy to ship and hard to spot. We separate hollow from canary by an explicit *alignment rule*[2] over three relations: the call graph $\text{calls}(f)$, the proof-dependency graph $\text{thmDeps}(T)$, and the *about* relation $\text{about}(T) = \{f \mid f \text{ appears in } T\text{'s type}\}$. A chain link (T, f)—a theorem T paired with a function $f \in \text{about}(T)$—*extends* to (T', f') when all three of the following hold simultaneously:

(i) $T' \in \text{thmDeps}(T)$ — T' is cited in T's proof term;
(ii) $f' \in \text{about}(T')$ — the type of T' mentions f';
(iii) $f' \in \text{calls}(f)$ — f's body calls f'.

The square with $T \to T'$ on top, $f \to f'$ on the bottom, and about arrows down both sides must commute. T *fibrates* when this rule iterates into a maximal chain—a discrete Grothendieck fibration of the proof DAG over the call DAG. A fibrating theorem is a *verification canary*: any behavioural change in the implementation invalidates a cited lemma, and the kernel rejects the proof. Architecture is therefore itself a requirements artifact—a flat design gives the audit nothing to mirror. `EndToEndCorrectness` ships two soundness headlines: `parse_sound_shallow` (type mentions only `parseYaml`; proof cites no project lemma: no fibration) and `parse_sound_deep` (type mentions every pipeline stage; proof composes `parseYamlRaw_ok_decompose` with `parseYaml_produces_valid_nodes`: deep fibration). Both check, but a scanner refactor that silently corrupts emitted content leaves the shallow one intact and breaks the deep one (Fig. 1). `theoremgraph -chain` classifies each capstone as *deep*/*propBridge*/*weak*; `check-capstones` fails the build on regression. The underlying structure is categorical (Fig. 2): traceability from formalized requirements to software architecture is a fibration functor whose cartesian-pullback lifting is requirement propagation, and the alignment rule is the same statement at the level of individual theorems.

Categorical foundation and a literature gap. The fibration's orientation matters: the *base* is the software architecture (functions, call graph) and the *total* is the requirements space (formal requirements, AND/OR dependencies, proofs of discharge). The pullback test confirms it: if A calls B and B has a requirement, the cartesian lift yields the weakest precondition A must enforce—Hoare-style verification. Reversing fails (a stronger requirement does not synthesise better software), in line with Lawvere's rule that predicates are fibered

[2] See `https://nasa-jpl.github.io/L4YAML/Verification/Mind-the-Fibration-Gap/#alignment-rule`.

over contexts [10]. Fibrations are well established in categorical logic [11] and in compositional models (optics, lenses [12]); but, to our knowledge, pairing a call graph (base) with a requirements/proof-discharge category (total) is absent from both literatures—the Fibration Gap Metric is a first mechanisable instance.

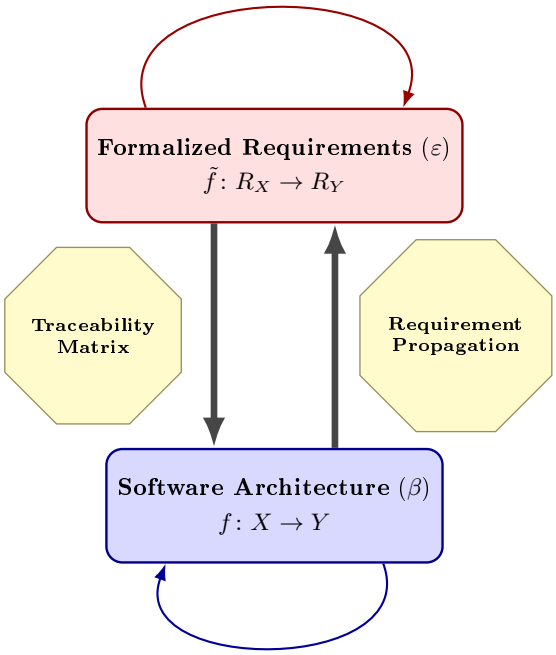
5 Conclusion

Each failed proof revealed a requirement (state restoration, context-free character classification, matched brackets, predicate polarity) that the prose never stated and no test exercised: requirements about *relationships between definitions*, not I/O behaviour. Four contributions: (1) *proof as requirements discovery*; (2) *five proof-breaking code patterns* generalising beyond parsers; (3) *a four-role Gen-AI workflow* realised through distinct prompting strategies; (4) *the Fibration Gap Metric*, a kernel-objective audit that separates verification canaries from hollow theorems—essential when Gen-AI generates both code and proofs. Future work feeds the bridging theorems' dependency graphs into proof blueprints [9] for auto-generated assurance cases.

Acknowledgments. Work performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with NASA (80NM0018D0004); Gen-AI assistance via Claude Code in VS Code.

References

1. Ben-Kiki, O., Evans, C., döt Net, I.: YAML 1.2.2, `yaml.org/spec/1.2.2/` (2021)
2. yaml-test-suite, `github.com/yaml/yaml-test-suite` (2024)
3. Rouquette, N.F.: L4YAML, `github.com/nasa-jpl/L4YAML` (2026)
4. Dorais, F.G.: lean4-parser, `github.com/fgdorais/lean4-parser` (2026)
5. Wadler, P.: Theorems for free! In: FPCA '89, pp. 347–359. ACM (1989)
6. MITRE: CWE-776 (XML entity expansion), on `cwe.mitre.org`
7. Rouquette, N.F.: `doc-verification-bridge` report about L4YAML, on `https://nicolasrouquette.github.io/doc-verification-bridge/L4YAML/site/`
8. Lowe, E.J.: The Four-Category Ontology. Oxford University Press (2006)
9. Massot, P.: LeanBlueprint, `https://github.com/PatrickMassot/leanblueprint` (2024)
10. Lawvere, F.W.: Equality in hyperdoctrines and comprehension schema as an adjoint functor. In: Heller, A. (ed.) Applications of Categorical Algebra. Proc. Symposia in Pure Math. XVII, pp. 1–14. AMS (1970)
11. Jacobs, B.: Categorical Logic and Type Theory. Studies in Logic and the Foundations of Mathematics, vol. 141. Elsevier (1999)
12. Capucci, M., Gavranović, B., Malik, A., Rios, F., Weinberger, J.: On a fibrational construction for optics, lenses, and Dialectica categories. Electronic Notes in Theoretical Informatics and Computer Science 4 (2024). `doi:10.46298/entics.14638`



Total Category ε (Formalized Requirements)
Objects = formalized requirements (e.g. R_X, R_Y)
Morphisms = logical dependencies, e.g. $\tilde{f}$ proving R_X satisfies R_Y's assume/guarantee

Traceability $\varepsilon \to \beta$ is a fibration functor
maps each requirement to the components/functions that satisfy it (X must enforce R_X's contract)

Propagation is its cartesian pullback
Base morphism $f\colon X \to Y$
Total object R_Y
Pullback verify R_X ensures the assumptions of R_Y

Base Category β (Software Architecture)
Objects = components / functions (e.g. X, Y)
Morphisms = interfaces / call graph (e.g. X calls Y)

Fig. 2. Putting it all together: requirements fibrate to functions via theorems. The total category ε has formalized requirements as objects and proofs of assume/guarantee discharge as morphisms; the base category β has functions as objects and the call graph as morphisms. Traceability is a fibration functor $\varepsilon \to \beta$, and requirement propagation is its cartesian-pullback lifting (the weakest-precondition). The *alignment rule* of §4 is the same statement at the level of individual theorems: a deep capstone such as `parse_sound_deep` is a chain of ε-morphisms whose image under traceability is the call chain that the theorem is about.