Super-intelligence Survival Guide: Verification via Proof-Carrying Output

Hillel Avni
Ben-Gurion University of the Negev, Israel
hillel.avni@gmail.com

Shlomi Dolev
Ben-Gurion University of the Negev, Israel
dolev@bgu.ac.il

Avraam Yagudaev
Ben-Gurion University of the Negev, Israel
avraamy@post.bgu.ac.il

Moti Yung
Google, USA
motiyung@gmail.com

Abstract

The increasing deployment of *large language models* (LLMs) in high-stakes domains demands infrastructure to ensure trust in *artificial intelligence* (AI)-generated outputs and actions. Users often struggle to validate results from LLMs because their reasoning is opaque and possibly beyond human comprehension. This paper introduces *proof-carrying output* (PCO), a framework in which an AI system returns an answer accompanied by a machine-checkable proof. We define φ-*compliance* formally (Definition 1): given a decidable predicate φ over signed inputs and AI outputs published by a named authority, a pair (x,y) is φ-compliant iff $\varphi(x,y) = 1$. "Compliance" in the rest of the paper refers to this binary, machine-checkable relation, not to organizational assurance practice. The framework is an instance of the producer-verifier-with-audit pattern previously introduced for game-theoretic rational behavior, applied here to regulatory compliance for AI-mediated decisions. Our primary security contribution is a *cryptographic accountability layer* that binds an AI output, its formal proof, the verifying validator's version, and a trusted timestamp into a non-repudiable commitment recorded on an append-only ledger *before* the output is acted upon. This layer provides four properties—binding, hiding, temporal ordering, and audit correctness—which jointly yield non-repudiation of AI-mediated decisions, a property neither LLM outputs nor formal proofs provide in isolation. These proofs rely on established proof assistants such as Rocq (Coq)[1] for symbolic reasoning and (*linear temporal logic* (LTL), *signal temporal logic* (STL)) for temporal logics prior to output usage. A legal entity—whether a human subject to law or an AI agent bound by smart contracts—must employ independent proof validators to confirm that the inputs (multiple-choice selections and signed documents) correctly lead to the output under published specifications and regulations. After validation and before acting on the output, the legal entity cryptographically commits the query, specification, output, validator version, timestamp, and proof to an append-only ledger. The entity then proceeds based on the output and reveals the proof only during the audit. We demonstrate PCO through three case studies with working Rocq/STL implementations: tax computation, autonomous-vehicle compliance, and recommendation transparency, extending *proof-carrying code* (PCC) from static programs to dynamic AI outputs. To enable reliable proof generation as an enabling substrate, we propose that regulatory authorities publish specification-coupled *small language models* (SLMs) trained on canonical scenario-proof pairs; we view this as supporting infrastructure rather than the core security contribution. We explicitly delimit PCO's scope to compliance predicates expressible as decidable first-order logic with bounded quantification or STL over finite-horizon signals; free-form prose, input authenticity, and specification correctness are out of scope and treated as orthogonal problems. PCO complements existing approaches to interpretable and explainable AI by providing machine-verifiable certificates of compliance rather than human-readable rationalizations.

1 Introduction

LLMs generate answers faster than humans can verify them. Humans assisted by LLM explanations work more efficiently but become over-reliant, achieving below-baseline accuracy when explanations are wrong—especially when those explanations appear convincing [48]. Recent benchmarks demonstrate that even the best available LLMs detect errors in LLM responses at much lower accuracy and recall than humans, rendering them unreliable as general-purpose error detectors [24]. The problem worsens because people often over-rely on AI recommendations and require explicit cognitive forcing interventions to shift from trusting to thinking [8]. The gap between model capabilities and human verification widens as systems improve.

We treat LLM errors as adversarial. Outputs fail for mundane reasons—insufficient training data, optimization arti-

[1]The Coq proof assistant was recently renamed to Rocq. We use "Rocq" throughout this paper, with "Coq" appearing in contexts where the historical name is more recognizable (e.g., "Coq verification time" in figures). Both names refer to the same proof assistant.

facts, architectural limitations—but they might also fail strategically. Morris et al. [35] propose a taxonomy of artificial general intelligence levels acknowledging that advanced systems may possess dangerous capabilities, including deception. That risk, however remote, motivates a strict position: if an AI makes a claim, it must prove it in a form a machine can check. The PCO usage in the scope of AI outputs shifts the trust problem from "trust the AI" to "trust the specification": rather than attempting to detect whether an AI is being deceptive, we require verifiable evidence that outputs satisfy published formal requirements.

The urgency of such infrastructure is underscored by recent strategic analyses. Hendrycks, Schmidt, and Wang [22] argue that "superintelligence—AI vastly better than humans at nearly all cognitive tasks—is now anticipated by AI researchers", necessitating a coherent strategy analogous to nuclear deterrence. Their framework of *Mutual Assured AI Malfunction* (MAIM) addresses geopolitical coordination, but does not address the accountability gap for AI systems operating within sanctioned development: how do we verify that a lawful AI system actually complies with its stated specifications? PCO provides this missing layer. While MAIM proposes deterrence against destabilizing AI development, PCO enables verification that deployed systems satisfy formal compliance requirements—shifting from trust in institutions to trust in proofs.

This need intensifies as AI systems exhibit *emergent abilities*—capabilities that appear unpredictably at scale [6, 51]. Anderson's principle that *more is different* [2] manifests in language models as qualitative capability jumps that resist prediction from smaller-scale behavior. Recent surveys document emergent phenomena ranging from advanced reasoning to in-context learning, alongside emergent risks including deception and harmful content synthesis [6]. [37] The connection to PCO is the following: because new failure modes can emerge without warning, accountability infrastructure should not depend on enumerating known failures in advance. PCO addresses this uncertainty not by predicting what will emerge, but by requiring that whatever emerges must satisfy formal specifications—converting unpredictable capability into auditable compliance for the predicate families in scope.

Necula's PCC [38] established this principle for untrusted programs. The code producer supplies the proof; the consumer only verifies. Verification costs almost nothing; proof construction bears the computational burden. Trust reduces to a small, auditable checker. Komendantskaya [26] recently extended PCC to neuro-symbolic systems, developing languages such as Vehicle that interface with neural network verifiers and proof assistants. Researchers have made substantial progress on certified proof checkers for neural network properties, but LLM-scale verification remains an open challenge.

We adapt PCC to generative AI outputs. The AI proves its answer satisfies a formal specification – but the model weights, training data, and proprietary algorithms remain hidden. PCO uses cryptographic commitments for privacy-preserving audit capability, distinct from zero-knowledge proofs: commitments hide the proof during normal operation, but the correctness proof for the output (rather than the techniques/logic used to obtain the output) is disclosed during authorized audits. This approach provides selective disclosure similar to commitment schemes: the proof remains hidden until it is selected for a warranted audit, but in this stage, unlike zero-knowledge proofs, auditors eventually see the full proof to verify compliance. The proofs may include domain-specific user inputs (e.g., tax documents, vehicle sensor readings), which stay private until audit. Model cards [34], datasheets [18], and audit frameworks [43] provide necessary complements but have fundamental limitations: dataset creators cannot anticipate every possible use, identifying unwanted biases often requires demographic labels that may be unavailable [18], and model cards seem unlikely to be standardized or formalized to a degree needed to prevent misleading representations [34]. We shift primary trust to reputable, open-source (small) proof checkers whose correctness can be independently verified by the public. The proof checkers use pure logic, using agreed-upon axioms and specifications globally.

Proofs may contain sensitive information, so public disclosure is undesirable. Our solution uses cryptographic commitments: the legal entity (human user or AI agent) receives the output and proof from the AI system, employs independent validators to verify correctness, and upon validation approval, cryptographically commits the query, specification, output, proof, and validator information before acting on the output. We note that the verification can be done using secure multi party computation, say, executed over blockchain servers. This binding step prevents retroactive changes and ensures that no one can later claim the proof addressed a different query. The proof remains private until an auditor requests it. Only the hash h appears on the public ledger; the proof itself, if revealed at all, goes only to authorized auditors.

System Overview. The PCO system operates as follows: a client sends a query to an LLM, which generates both an answer and a formal proof demonstrating correctness with respect to published formal regulations. The client may use digitally signed documents (e.g., bank statements, medical records, certificates) to accompany the query. The client receives this proof-carrying output and employs independent validators to verify it locally. Each validator signed the verification result. Upon successful validation, the client locally stores all signed documents, along with the complete proof, and commits the entire record to an immutable blockchain ledger using cryptographic hashes. Then the client can safely choose to act according to the LLM output. In this framework, a regulatory authority serves two critical functions: first, it establishes and publishes the formal specifications and legal requirements that govern admissible behavior; second, it

conducts audits when needed, requesting the record containing the full proof from the client to verify compliance. This design ensures that routine operations remain private while preserving accountability through selective disclosure during regulatory oversight.

Contributions. PCO pairs each AI answer with a proof that the machine can check. A cryptographic commitment locks them together. The query and proof are bound at commit time, with no later swapping of specifications (Section 3). We prove four security properties—binding, hiding, temporal ordering, and audit correctness—under standard assumptions (Section 4).

Primary Security Contribution. We explicitly position the paper's security contribution as the *cryptographic accountability layer*, not the proof-generation pipeline. Proof generation (Section 5.1) is the *enabling substrate*; the security contribution is the protocol that binds outputs, proofs, validator identities, and timestamps into a non-repudiable audit trail before action. This layer provides four properties that neither raw LLM outputs nor standalone formal proofs supply: (i) *commitment binding* (an output cannot be retroactively reattributed to a different specification or query), (ii) *hiding* (the proof remains private until a warranted audit), (iii) *temporal ordering* (action provably follows validation), and (iv) *audit correctness* (the committed evidence is verifiable years later). Together these yield *non-repudiation of AI-mediated decisions*: a property current model cards, watermarking schemes, signed logs, and zero-knowledge ML systems do not jointly achieve.

Scope of Tasks and Predicates. We make our scope explicit. PCO targets tasks whose outputs are *structured objects* for which correctness is *deterministically re-derivable* from signed inputs under a published predicate. Concretely, we handle three predicate families exemplified by our case studies:

- **Arithmetic-functional predicates** (tax computation): bracket selection, deduction eligibility, and refund derivation, expressed as decidable first-order propositions over signed financial documents.
- **Temporal-safety predicates** (autonomous vehicles): bounded-horizon STL formulas over sensor signals (e.g., $\square_{[0,T]}(\mathit{speed} \leq \mathit{limit} \wedge \mathit{gap} \geq \mathit{required})$).
- **Ranking/optimality predicates** (recommendations): utility-maximization and disclosure requirements, expressed as universally quantified comparisons over a certified product catalog.

Common to all three is that $\varphi(x,y)$ is decidable in polynomial time given x and y, uses bounded quantification, and admits machine-checkable proofs of fixed structural form. *Out of scope*: free-form natural-language outputs, predicates over unbounded data, properties requiring full higher-order reasoning, and the correctness of inputs x themselves.

Scope and Current Capabilities. We present PCO as infrastructure for AI accountability, demonstrating feasibility with contemporary LLMs on representative compliance scenarios. Our experimental evaluation (Section 5.4) employs *Retrieval-Augmented Generation* (RAG) [29] to enhance LLM proof generation with domain-specific patterns and error recovery strategies. The RAG-enhanced pipeline achieves reliable proof generation across all tested scenarios by providing LLMs with: (1) working code examples from the target domain, (2) syntax rules specific to the formal language, and (3) error-specific correction hints when initial attempts require refinement. We propose that regulatory authorities publish *specification-coupled proof generators*—combining RAG knowledge bases with SLMs fine-tuned on canonical (scenario, proof) pairs derived from their specifications. For the predicate families in our scope (Section 1), this combination reduces proof generation from an open research problem to a bounded engineering task with measurable success rates. We emphasize that PCO provides value today for the compliance rules that can be expressed within the predicate families above, while establishing architectural patterns that may extend to future capabilities. We do not claim that arbitrary regulations can be formalized; formalizing law remains a hard, open problem (Section 5.2), and we tone down any earlier suggestion to the contrary.

All developments for the three case studies, including full proof scripts and build scripts, are available in our repository: https://github.com/hillelav/pcoq.

Organization. Section 2 surveys related work. Section 3 presents the protocol. Section 4 provides security analysis. Section 5 discusses anticipations and limitations, followed by concluding remarks and full case study specifications.

2 Related Work

2.1 Proof-Carrying Code

In PCC [38], whoever writes the code must also supply the proof; the person running it only verifies. Necula targeted untrusted mobile code in operating systems. The code ships with a safety proof, and the host checks it against a verification condition generator. Verification runs in time linear in proof size. Komendantskaya [26] brings PCC into neuro-symbolic territory, requiring neural controllers to be accompanied by safety proofs. This line of work often combines with tools such as Vehicle, which interface neural-network verifiers with proof assistants. LLM-scale verification with foundational guarantees has not yet been achieved.

PCC vs. PCO Trust Model. A key distinction: PCC verifies code before execution, while PCO verifies outputs after generation but before action. Both share the core principle that the producer bears the burden of proof and the consumer only verifies. In PCO, verification precedes *action* on the output—the legal entity commits the proof before acting on the AI's recommendation. This preserves the "verify-then-act" paradigm central to PCC's security guarantees. The trust invariant remains the same: nothing is trusted without machine-checkable evidence. The exact contract being verified—φ-compliance, Definition 1—is stated formally in Section 3.1, which also explains why the Rocq-over-LLM composition is a principled producer-verifier separation rather than an opportunistic engineering stack.

What is fundamentally different about generative AI. Three issues separate PCO from classical PCC and from neural-network property verification: (i) *Subject shift.* PCC verifies fixed code against state-machine properties known at compile time. PCO verifies a stochastic input-output pair (x, y), where y is freshly sampled from a learned distribution. The specification must therefore jointly quantify over inputs and outputs, not just over program states. (ii) *Per-query proof search.* PCC proofs are constructed once at compile time and shipped with the binary. In PCO, a new proof is generated for every query at inference time, introducing a latency-success trade-off that PCC does not face. This makes proof-search engineering (RAG, SLMs, fallback templates) a first-class system component rather than a one-shot static-analysis artifact. (iii) *Bounded output space, unbounded reasoning.* LLM output tokens are drawn from a finite vocabulary. The challenge is not the alphabet size; it is that the predicate $\varphi(x, y)$ must be decidable for whatever well-typed y the model produces, and that the supporting proof must compile in a chosen logic. Neural-network verification (e.g., Vehicle, Marabou) addresses input-output relations of a fixed neural function. PCO instead asks: *regardless of how y was produced, does (x, y) satisfy a published φ?* This decouples accountability from model internals and is what makes PCO compatible with closed-weight commercial LLMs.

2.2 Rationality Authority and Proof-Verified Behavior

The closest intellectual antecedent of PCO is the *Rationality Authority* (RA) framework of Dolev, Panagopoulou, Rabie, Schiller, and Spirakis [14]. RA targets distributed game-theoretic settings where players must take utility-optimal actions but their utility functions are private. The framework provides each player with a procedure that (i) computes the optimal action under the player's hidden utility, and (ii) outputs a machine-checkable *proof of optimality* that can be verified by many reputable proof checkers without revealing the utility itself. Auditing ensures that a player actually used the provided procedure: the player is required to keep signed information about the choice, available for future inspection.

PCO is an instantiation of the same architectural pattern in a different domain. The mapping is direct:

- *RA's optimality predicate* ("action a maximizes utility u over the action set") becomes PCO's *compliance predicate* $\varphi(x, y)$ ("output y satisfies the published regulatory predicate over context x," Definition 1).
- *RA's procedure with a private utility* becomes PCO's LLM-based proof-generation pipeline with a private model and private inputs.
- *RA's reputable proof checkers* become PCO's independent validators running Rocq or STL monitors (Section 5.3).
- *RA's audit-by-signed-information* becomes PCO's commitment to an append-only ledger and selective disclosure (Section 4).

Two extensions distinguish PCO from RA. First, the predicate originates with a regulatory authority rather than with the player, which shifts the locus of correctness from individual rationality to public policy. Second, the proof producer is a learned, stochastic LLM rather than a deterministic computation, which is what motivates the producer-verifier separation and the failure-mode analysis of Section 5.1. The cryptographic accountability layer of Section 4 adapts RA's audit-trail discipline to the post-quantum, non-repudiable setting of AI-mediated decisions. The point of including this lineage explicitly is that PCO is not a free-standing engineering proposal; it is the latest point on a research arc that started with proof-verified *rational* behavior and is being extended here to proof-verified *compliant* behavior.

2.3 LLM Proof Generation

LLMs can now generate formal proofs. Autoformalization experiments demonstrate that LLMs perfectly formalize a significant fraction of competition mathematics problems into Isabelle/HOL, and that the resulting data measurably improves neural theorem provers [53]. Given the rate of LLM improvements, we anticipate that the fraction of useful use cases for which LLMs will be able to produce (independent rather than operational) proofs of their outcomes will include almost all scenarios. LLEMMA [3] and DeepSeek-Prover [54] show that models address involved mathematical statements. We treat the LLM as an untrusted prover in a cryptographic protocol: it may produce correct proofs, but the verifier assumes nothing about its behavior. The potential for unintended behavior in AI systems [1, 46] motivates treating the LLM as untrusted.

2.4 Cryptographic Primitives

Our protocol requires binding proofs to their originating queries before the AI acts. Hash-based commitments e.g., [40] tie queries, specifications, outputs, and proofs into a single bundle. To commit, the system hashes together the query q, specification φ, output y, proof π, and a fresh nonce r, producing $h \leftarrow \mathsf{H}(q \,\|\, \varphi \,\|\, y \,\|\, \pi \,\|\, r)$. The hash h then goes to an append-only ledger. We encode tuples prefix-free to avoid ambiguity when concatenating. Because finding hash collisions is hard, the system cannot later pretend it proved φ when it actually proved φ', or claim it answered query q' instead of q. The nonce keeps everything hidden until the commitment opens.

Post-quantum security protects *commitment integrity*, not proof validity. Rocq verification is sound regardless of cryptographic assumptions; the proof checker's correctness is independent of the computational power of an adversary. However, the commitment binding the query to the proof must remain secure over regulatory timeframes. Audit trails for tax filings, medical recommendations, or autonomous vehicle decisions may span 10–30 years, well into an era when quantum computers could break classical cryptographic assumptions. Hash-based commitments using SHA-256 achieve post-quantum security because Grover's algorithm [21] provides only a quadratic speedup against collision resistance, yielding approximately 128-bit security against quantum adversaries.[2] The prover commits before execution, preventing the proof from being adapted to the verifier's challenge. We apply the same logic to AI accountability: the system commits its proof before delivering the output to the user, preventing retroactive justification.

Post-Quantum Soundness of Proof Systems. Beyond commitment integrity, one may ask whether the underlying proof systems (Rocq, STL/RTAMT) remain sound under quantum attacks. The short answer is yes, with one caveat:

- *Rocq's logical soundness* relies on the consistency of the Calculus of Inductive Constructions, a question of mathematical logic that is independent of computational hardness; quantum computers cannot "break" an inconsistent foundation, only solve search problems faster. Rocq proof checking is also deterministic and uses no cryptographic assumption.
- *STL monitoring* (RTAMT) is similarly a deterministic decision procedure over real-valued signals; no cryptographic assumption is involved.
- The single quantum-relevant component is the *commitment binding* and any *digital signatures* on x, t_s, or v. SHA-256 commitments retain $\approx$ 128-bit security under Grover; classical signature schemes (RSA, ECDSA) do not, and should be replaced with NIST-standardized PQC primitives (e.g., ML-DSA [36]) for long-lived audit trails.

The protocol is therefore post-quantum modulo the standard advice to migrate signing keys.

Sometimes, regulators need to see proof that the public should not. In that case, store proofs encrypted and escrow the keys to auditors. For query privacy, encrypt the query first: $h = \mathsf{H}(\mathsf{Enc}_k(q) \| \varphi \| y \| \pi \| r)$. Auditors who hold key k can still check the binding.

2.5 Runtime Verification

Autonomous systems require specifications over continuous signals and discrete events. LTL [42] handles discrete temporal properties with operators $\Box$ (always), $\Diamond$ (eventually), and U (until). STL [32] extends LTL to real-valued signals with explicit time bounds. Runtime monitors check these specifications on the fly—something that theorem provers cannot do; tools such as RTAMT [39] provide efficient quantitative monitoring with robustness semantics. Runtime monitoring alone, however, leaves an accountability gap: it tells us *whether* a property held at a moment, but not *why* the system chose its action, nor does it bind that record to a tamper-resistant audit trail. Koopman and Wagner [27] document precisely this gap: today's event recorders log sensor readings and actuator commands (*what* happened) but not the reasoning behind each decision, and rare "black swan" events plus overfitting make post-hoc reconstruction unreliable. PCO fills that gap by pairing the monitored property with a committed proof and signed timestamp. Each decision comes with a certificate proving it met the published spec, recorded on an append-only ledger before the action is taken.

2.6 Proof-Carrying Data

Chiesa and Tromer's *Proof-Carrying Data* (PCD) [10] is a cryptographic cousin of PCC: every message in a distributed computation carries a short proof that some global invariant still holds. Proofs in PCD stack: each step in a distributed computation can prove that it inherited a valid state and preserved the invariant.

Succinct non-interactive arguments of knowledge (SNARKs), such as *Pinocchio* [41] and later systems [25] compile arbitrary programs—and even large neural networks—into circuits for which a prover can generate short proofs of correct execution. Solatorio's *Proof-Carrying Numbers* (PCN) [49] tackles LLM outputs directly: each numeric answer comes with a machine-checked proof linking it to an authoritative source. PCO has more in common with PCC and PCN than with SNARK-based systems. Where SNARKs compile to arithmetic circuits, we stick to domain logics.

[2] For audit trails exceeding 30 years, 128-bit security may be insufficient given uncertainty about quantum speedups for generic collision-finding. The commitment scheme is modular: organizations requiring longer-term security can substitute SHA-3 or future post-quantum hash functions. The protocol design is hash-agnostic.

Moreover, we go beyond numbers to structured outputs: tax forms, vehicle maneuvers, and recommendations. We layer in cryptographic commitments for temporal ordering and selective disclosure.

We note that a direct experimental comparison with SNARK-based systems (e.g., Pinocchio, zkML [9]) or with verifiable computation frameworks (e.g., Cairo) is of limited relevance to our setting. These systems excel at proving correct execution of *fixed programs* over *private inputs*—a different problem than proving compliance of *AI-generated outputs* with *domain-specific regulations*. SNARKs require compiling specifications to arithmetic circuits, which is impractical for the rich logical predicates (quantifiers, inductive types, recursive definitions) needed in our case studies. Moreover, blockchain smart contract verification does not require the diversity of proof types that PCO demands: a token transfer is verified once against a fixed contract, whereas PCO must verify novel outputs against evolving specifications across heterogeneous domains. The architectural differences – circuit-based vs. logic-based, fixed-program vs. dynamic-output, single-domain vs. multi-domain – make comparisons less meaningful than demonstrating PCO's feasibility for its intended use cases.

Comparison Summary. Table 1 summarizes the key distinctions between PCO and SNARK/zkML approaches.

Table 1: Comparison of PCO with SNARK-based approaches

Dimension	PCO (Ours)	SNARKs/zkML
Proof target	Specification compliance	Program execution
Logic expressiveness	Full first-order + induction	Arithmetic circuits
Program flexibility	Dynamic outputs	Fixed programs
Privacy model	Selective disclosure	Zero-knowledge
Proof size	Linear in spec	Succinct (constant)
Verification time	Linear	Constant
Setup requirements	None (transparent)	Trusted setup*
Domain generality	Multi-domain	Single-circuit

*Some SNARKs (e.g., STARKs) avoid trusted setup.

The trade-offs reflect different design goals: SNARKs optimize for succinct, zero-knowledge proofs of fixed computations, while PCO optimizes for expressive, transparent proofs of specification compliance. For AI accountability, expressiveness and transparency outweigh succinctness, as specifications involve complex logical predicates and auditors benefit from inspecting full proofs.

2.7 Verifiable Computation

Languages for verifiable computation, such as Cairo, provide a practical foundation for executing programs together with succinct cryptographic proofs of correct execution. Cairo is designed to generate STARK proofs attesting that a computation was carried out faithfully according to its program semantics, without revealing intermediate state or private inputs [5, 19]. This approach has been explored in the context of scalable blockchains, privacy-preserving computation, and verifiable off-chain execution.

Such proof-oriented execution languages complement traditional formal methods. While proof assistants establish correctness at the level of logical reasoning, verifiable execution systems enable cryptographic enforcement of selected computational steps at runtime. Prior work has shown that this separation supports scalable verification in settings where executors and verifiers do not fully trust each other. These ideas motivate the use of zero-knowledge execution as an implementation substrate for systems that require auditability and privacy, including compliance-oriented pipelines.

2.8 Executable Specifications

Smart contract languages such as Solidity provide an influential precedent for eliminating ambiguity in executable specifications. Solidity programs encode contractual rules as deterministic state-transition systems enforced by the Ethereum Virtual Machine (EVM), yielding a single operational interpretation for each contract [17, 52]. Unlike natural-language contracts or policies, Solidity programs admit no double meaning: permissions, obligations, and constraints are defined entirely by execution semantics.

This shift from descriptive to executable specifications has been central to the deployment of decentralized systems involving mutually distrustful participants. Ambiguities that would otherwise require legal or human interpretation are resolved by re-executing code under a well-defined virtual machine. Prior work on smart contracts thus demonstrates how formalized, machine-executable specifications can replace intent-based interpretation with deterministic enforcement.

The PCO framework draws inspiration from this line of work by applying the same principle to compliance verification. Rather than encoding transactions, PCO encodes admissible decisions as formal predicates with unambiguous semantics. In this sense, PCO generalizes the idea of executable contracts from transactional correctness to decision correctness, leveraging both formal specifications and verifiable execution techniques as complementary tools.

2.9 Interpretable and Explainable AI

The fields of *Interpretable AI* (IAI) and *Explainable AI* (XAI) seek to make AI system behavior comprehensible to humans [4, 15, 30]. Interpretable models (e.g., decision trees, linear models, rule lists) are designed to be inherently transparent, while explainable systems generate post-hoc rationalizations for opaque models (e.g., LIME [44], SHAP [31], attention visualization). Both approaches aim to answer *why* a model produced a particular output.

PCO addresses a complementary but distinct problem: rather than explaining *why* an AI reached a conclusion, PCO proves *that* the conclusion satisfies a formal specification. This distinction has important implications:

Rationality vs. Rationalization. XAI explanations are often post-hoc rationalizations that may not reflect the model's actual computation [45]. Attention weights, for example, correlate poorly with feature importance in many architectures [23]. In contrast, PCO proofs are logically sound: if the proof verifies, the output *necessarily* satisfies the specification, regardless of how the model internally arrived at it. PCO thus provides *rationality guarantees* (the output is provably correct) rather than *rationalization artifacts* (the output seems reasonable).

Machine vs. Human Verification. XAI targets human comprehension, assuming that if humans understand the reasoning, they can assess correctness. This assumption fails when: (1) explanations are too complex for human evaluation, (2) humans over-trust plausible-sounding explanations [8,48], or (3) the model's reasoning is fundamentally alien to human cognition. PCO sidesteps these limitations by targeting machine verification: the proof checker requires no human judgment and scales to arbitrary complexity.

Complementary Roles. PCO and XAI serve different stakeholders. XAI helps developers debug models, users build appropriate trust, and regulators assess fairness heuristically. PCO provides cryptographic accountability for high-stakes decisions where informal explanations are insufficient. A deployed system might use XAI for routine user-facing explanations while reserving PCO for audit-critical compliance verification.

We note that PCO proofs can themselves serve as a form of rigorous explanation: the proof structure reveals which specification clauses the output satisfies and how input data maps to the conclusion. Unlike natural-language explanations, these formal explanations are complete (covering all verification conditions) and verifiable (independently checkable). Future work could explore "proof summarization" techniques that translate formal proofs into human-readable explanations, combining the rigor of PCO with the accessibility of XAI.

3 Protocol

We extend PCC-style proof-carrying ideas to generative AI.

An LLM produces an answer y together with a formal proof π in an established logical system—Rocq [7] for symbolic reasoning, temporal logic for execution traces. The legal entity receives the output and proof, employs validators to check compliance, and upon approval commits the bundle to the ledger before acting.

We make this precise by first defining the property under verification (Section 3.1), then specifying the protocol's algorithms (Section 3.2), and finally stating its security guarantees (Section 4). The sequence—property, mechanism, guarantee—mirrors the structure of standard cryptographic protocol specifications and is intentional: it eliminates the "ad-hoc combination" reading in which Rocq and an LLM are simply bolted together.

3.1 Compliance: Formal Definition and Motivation

"Compliance" is used in this paper before being formally defined, which can make the Rocq-over-LLM construction look like an ad-hoc engineering combination. We rectify this by defining compliance formally and showing that, given the definition, the protocol is a standard producer-verifier separation rather than an opportunistic stack.

Informal Notion vs. Our Use. In everyday regulatory practice, "compliance" denotes an organizational property: an entity files paperwork, runs controls, signs attestations, and is audited periodically. The artifact is a process, not a binary judgment on a specific output. None of this yields a machine-checkable witness binding a specific AI output to a specific rule. Our use of the term is narrower and machine-oriented; we mean a property of the pair (x,y), decidable in bounded time against a published predicate.

Definition 1 (φ-Compliance)**.** Let X be a space of admissible inputs and Y a space of admissible outputs, both representable as finite bit-strings. Let $\varphi : X \times Y \to \{0,1\}$ be a *decidable, total predicate* that (i) is published by a named regulatory authority with a stable identifier, (ii) admits a finite syntactic representation in a chosen logic (Rocq specifications for first-order properties; STL formulas for real-valued temporal signals), and (iii) is computable in time polynomial in $|x|+|y|$. A pair $(x,y) \in X \times Y$ is φ-*compliant* iff $\varphi(x,y) = 1$.

Definition 2 (Verifiable Compliance)**.** An AI system $M : X \to Y$ is *verifiably* φ-*compliant on* x iff there exists a witness π and a verification algorithm $\mathsf{Verify}(\varphi, x, y, \pi)$ such that $\mathsf{Verify}(\varphi, x, M(x), \pi) = 1$ implies $\varphi(x, M(x)) = 1$. We require Verify to be sound, deterministic, and to run in time polynomial in $|\varphi| + |x| + |y| + |\pi|$.

Why this is the right notion for AI accountability. Four properties of Definition 1 make it usable as the substrate for the cryptographic accountability layer of Section 4:

1. *Decidability.* φ has a yes/no answer in bounded time. Goals like "the model is fair in some statistical sense" that cannot be evaluated from (x,y) alone are intentionally excluded; they are addressed by separate machinery and lie outside PCO's scope.
2. *Publication.* φ has a stable, citable, public representation (a Rocq file, an STL formula). Two parties can refer to

"the same φ" unambiguously and can include its hash in a commitment.

3. *Re-derivability.* Anyone holding (x, y, π) can re-run $\mathsf{Verify}(\varphi, x, y, \pi)$ years later, regardless of whether the AI that produced y still exists. Audit does not depend on model availability.
4. *Commitability.* Because φ, x, y, π are each strings of bounded length, their concatenation can be cryptographically committed at fixed cost, enabling the binding/hiding/temporal-ordering/audit-correctness properties of Section 4.

Why Rocq-over-LLM is not ad hoc. Given Definition 2, the choice of an LLM as producer and Rocq as verifier follows from a standard *producer-verifier separation* that originates with PCC [38] and was generalized to game-theoretic rational-behavior settings by Dolev et al.'s Rationality Authority [14] (see Section 2.2). The LLM samples candidate (y, π) pairs from an untrusted, learned distribution; Rocq decides $\mathsf{Verify}(\varphi, x, y, \pi)$ on a small, deterministic, audited codebase. Soundness of the composition reduces to soundness of Rocq, independent of the LLM. STL with a runtime monitor such as RTAMT [39] plays the same producer-verifier role when φ ranges over real-valued temporal signals, where a full Rocq proof would be too heavyweight for sub-millisecond decision rates (Section 10). The choice of logic is dictated by the syntactic shape of φ, not by engineering preference: Rocq for first-order arithmetic and inductive structures (tax, recommendations), STL for bounded-horizon properties of continuous signals (autonomous vehicles). Neither logic is novel; the contribution is the protocol that wraps them.

Boundaries of the definition. Definition 1 deliberately does not capture: (a) *specification correctness*—whether φ faithfully encodes the regulator's intent (Section 5.2); (b) *input authenticity*—whether the signed x truthfully describes the world (Section 4); (c) *counterfactual fairness*—whether M would have produced an equally compliant y' for a different x'. Each is a real and important problem; each is orthogonal to the question PCO answers. The definition lets us state precisely what PCO does and does not provide.

3.2 Syntax

A PCO scheme consists of five algorithms (Figure 1 shows their composition; we now make every argument explicit):

- $\mathsf{Generate}(q, \varphi, x) \to (y, \pi)$: On query q, specification φ, and context x, where x represents domain-specific inputs (tax documents for tax compliance, sensor traces for autonomous vehicles, or product catalogs for recommendations), the AI system produces output y and a proof π that $\varphi(x, y)$ holds. The generator has access to φ so that the proof targets the correct predicate—without this, no claim of compliance is meaningful.
- $\mathsf{Verify}(\varphi, x, y, \pi) \to \{0, 1\}$: Independent validators check whether π proves that (x, y) satisfies specification φ.
- $\mathsf{Commit}(q, \varphi, x, y, \pi, v, t_s) \to (h, r)$: The legal entity samples $r \leftarrow \{0,1\}^\lambda$ and computes $h \leftarrow \mathsf{H}(q \,\|\, \varphi \,\|\, x \,\|\, y \,\|\, \pi \,\|\, v \,\|\, t_s \,\|\, r)$, binding query, specification, context, output, proof, validator version v, and signed timestamp t_s together. Including x in the commitment was implicit in the previous version (carried by π, which references x); we now state it explicitly.
- $\mathsf{Audit}(h, q, \varphi, x, y, \pi, r, v, t_s) \to \{0, 1\}$: The auditor checks whether $\mathsf{H}(q \,\|\, \varphi \,\|\, x \,\|\, y \,\|\, \pi \,\|\, v \,\|\, t_s \,\|\, r) = h$ and re-runs $\mathsf{Verify}(\varphi, x, y, \pi)$.

Running Example. Throughout the rest of the paper we use the tax case study as a running example to ground these abstractions. There: q is "compute my 2024 federal tax return," φ is the IRS-published predicate `claim_is_correct` (Section 9), x is the bundle of signed W-2 and 1099 documents, y is the proposed return, π is the Rocq proof that $\varphi(x, y)$ holds, v identifies the validator binary, and t_s is the RFC 3161 timestamp obtained after validator approval.
The protocol flow is summarized in Figure 1.

3.3 Proof Scope and Specification Authority

PCO focuses on specifications that can be expressed in logical fragments, which are decidable and allow verification to complete in polynomial time. We do not claim that LLMs can prove arbitrary statements—computability theory establishes fundamental limits on provability. We focus on the practical side, as blockchain smart contracts do. The proof π confirms only that the output y satisfies the predicate $\varphi(x, y)$. A trusted proof checker handles verification.

The class of compliance rules that fits within PCO's scope (Section 1) is non-trivial but bounded. Our three case studies—tax computation, autonomous vehicle compliance, and recommendation transparency—demonstrate feasibility within this scope. We do not claim that the scope will grow to encompass arbitrary regulation; that depends on independent progress in legal formalization (Catala, Symboleo), formal methods, and LLM proof-generation capability, none of which is a contribution of this paper.

Given any input (q, x), the proof confirms that the output y satisfies the specification that the checker accepts (The commitment locks this proof to query and specification). However, out of the proof scope is: the LLM's internal reasoning, as we treat the model as a black box, optimal proof(unless φ explicitly requires it), and input authenticity, that is, if x is fraudulent, the proof still holds as fraud lies outside our threat model.

Proof-Carrying Output Protocol

1. **Query:** Legal entity sends query q to the AI system with specification identifier φ.
2. **Generation:** AI system computes $(y,\pi) \leftarrow \mathsf{Generate}(q,x)$.
3. **Delivery to Legal Entity:** AI system sends (y,π) to the legal entity.
4. **Validator Employment:** Legal entity employs independent validator(s) with version identifier v to check $\mathsf{Verify}(\varphi,x,y,\pi) = 1$.
5. **Timestamp Acquisition:** Upon validator approval, legal entity obtains signed timestamp t_s from trusted time service.
6. **Commitment:** Legal entity computes $(h,r) \leftarrow \mathsf{Commit}(q,\varphi,y,\pi,v,t_s)$ and publishes h to ledger $\mathcal{L}$ at time t_c.
7. **Action:** After time $t_e > t_c + \delta$ (where δ bounds clock drift), the legal entity acts according to output y.
8. **Regulatory Audit:** On later audit, the auditor retrieves h from $\mathcal{L}$, requests $(q,\varphi,y,\pi,r,v,t_s)$ from the legal entity, and checks $\mathsf{Audit}(h,q,\varphi,y,\pi,r,v,t_s) = 1$ and $\mathsf{Verify}(\varphi,x,y,\pi) = 1$.

Output structure: PCO = (y,π,h,r,v,t_s) where y is output, π is proof, $h = \mathsf{H}(q \,\|\, \varphi \,\|\, y \,\|\, \pi \,\|\, v \,\|\, t_s \,\|\, r)$, v is validator version, t_s is signed timestamp, and r is the nonce.
Query-proof binding: The commitment h cryptographically binds query q, specification φ, output y, proof π, validator version v, and timestamp t_s together, preventing the entity from later claiming that a valid proof for specification φ' was actually a proof for φ, or that it responded to a different query q'.
Temporal requirement: The delay $t_e > t_c + \delta$ ensures finalization of the commitment before action, preventing retroactive proof substitution. The legal entity must obtain a timestamp of commitment to the transaction and execute the action with a later timestamp, a signed timestamp accompanying the action invocation obtained by reliable time service providers, and kept for audit.
Context: The context x is domain-specific (user-provided documents, system-generated traces, or environmental data) and available to both the legal entity and the auditor.
Validator responsibility: The legal entity must employ reputable validators to verify proof correctness before commitment. The validator version v is committed to enable audit of the validation process.
Privacy: The proof π remains private and is revealed only to authorized auditors upon request, not at commitment time.

Figure 1: PCO protocol execution. Legal entities employ validators to verify binding and compliance before commitment; regulators verify temporal ordering via the ledger and proof of correctness upon audit.

Specifications must come from authorized authorities to ensure accuracy and reliability, rather than from users or AI systems (which may lack those two properties). The providing specification process is handled once by the appropriate authority, which publishes formal rules (i.e., for tax compliance, the IRS publishes formal predicates defining valid deductions). Section 5.2 discusses specification authorship and governance in detail.

The following are out of scope in this paper: responsibility allocation, usage policy, and liability. The PCO protocol provides forensic capability with the legal entity bearing responsibility for employing validators and making commitments.

4 Security Analysis

4.1 Threat Model

Actors and Attack Surface. The commitment in Figure 1 is performed exclusively by the legal entity, not by the AI provider. We therefore distinguish four classes of adversary: (i) an adversarial AI provider that may emit invalid proofs, (ii) an adversarial legal entity that may attempt to bypass validation or evade audit, (iii) an adversarial validator or auditor, and (iv) adversarial inputs x. "Proof substitution after commitment" is an attack on the legal entity (or on the ledger), not on the AI. We make each class explicit below.

We consider an adversarial AI system that may attempt to: (i) produce proofs for different statements than claimed, (ii) collude with the legal entity to enable post-commitment substitution attempts, (iii) backdate commitments (jointly with a corrupt time service), or (iv) deny having made com-

mitments.

We consider three adversarial scenarios:

Adversarial AI Provider: The AI system may attempt to generate invalid proofs, substitute proofs post-commitment, or collude with users to bypass validation. The protocol's cryptographic binding prevents post-commitment manipulation, but relies on validators to reject invalid proofs.

Layered Threat Model. We distinguish two classes of adversarial behavior: (1) *Protocol-level attacks*: generating invalid proofs, substituting proofs, backdating commitments—these are prevented by cryptographic binding and validator verification; (2) *Specification-level attacks*: generating valid proofs for harmful outputs that technically satisfy specifications—these are addressed at the specification governance level, not the protocol level. PCO's security guarantee is: "If an output is acted upon, it provably satisfies the published specification." The specification must be designed to exclude harmful outputs. This separation of concerns mirrors how cryptographic protocols provide security properties conditioned on correct key management: the protocol is secure, but key compromise is a distinct threat.

Indirect Injection on x (Out of Scope). A natural question is whether indirect prompt injection attacks on the context x are considered. We make our position explicit: PCO certifies that the output y satisfies $\varphi(x, y)$ for the *signed* x that was committed. If an adversary manipulates x (e.g., poisons a retrieved document, injects instructions into a sensor feed) but x is authentically signed and committed, PCO produces a valid audit trail that attributes the resulting output to that specific x. Liability then shifts to the party responsible for x's integrity (the document signer, the sensor manufacturer, the V2I issuer). This is consistent with how digital signatures handle adversarial documents: the signature certifies authorship, not benign intent.

PCO is designed to *compose additively* with input-authenticity mechanisms—hardware-attested sensors [50], signed regulatory documents (e.g., IRS-issued W-2 forms), V2I message authentication, and provenance ledgers—rather than replace them. Specifications can also *require* signed inputs (e.g., "each W-2 in x must carry a valid IRS employer signature"), making input authenticity verifiable *within* the specification itself. Designing prompt-injection-robust LLMs is an active research area orthogonal to ours [20, 55] and we do not claim to advance it.

Adversarial Legal Entity: A malicious user may attempt to: (a) act without proper validation, (b) employ compromised validators, (c) commit invalid proofs, or (d) refuse to reveal proofs during audit. The protocol addresses (a) and (c) through timestamp binding and validator version commitment. Cases (b) and (d) require validator reputation mechanisms and legal enforcement respectively (see Section 5.2).

Validator Ecosystem: We do not assume all validators are honest. Instead, we require legal entities to employ multiple independent validators or rely on validators with established reputation. A validator market (analogous to certificate authorities in PKI) provides economic incentives for honest verification. Auditors can verify which validator versions were used via the committed v.

Specification Capture. A central concern for any compliance framework is regulatory capture—the risk that regulated entities influence the formal specification φ to permit harmful behavior while maintaining formal compliance. PCO does not *prevent* capture; it *localizes and exposes* it. Three mechanisms apply:

1. *Specification Transparency Log.* Specifications are published to an append-only transparency log under the regulatory authority's signature, analogous to Certificate Transparency [28]. Any deviation between the published φ and the stated regulatory intent becomes a publicly observable artifact attributable to a named authority.
2. *Regulatory Sandboxing.* An independent oversight body (e.g., a legislative auditor or specification standards board) ratifies φ against scenario tests before enactment, raising the cost of capture from private negotiation to public falsification.
3. *Open Scenario Corpora.* Because (scenario, proof) corpora are public (Section 5.1), independent parties can submit adversarial scenarios that expose specification gaps; revealing capture-friendly clauses is therefore a positive-sum activity for civil society.

PCO thus reduces capture from an undetectable to a detectable problem; we do not claim to solve the political economy of regulation.

We assume: (1) the ledger $\mathcal{L}$ is append-only with trusted timestamps, (2) auditors are honest and computationally bounded, (3) the hash function H is collision-resistant, and (4) the (majority of the employed) Rocq/LTL proof checker correctly implements its verification algorithm.

Note on Auditor Trust. Assumption (2) merits elaboration. We assume auditors faithfully execute verification—they do not collude with legal entities to approve invalid proofs or selectively audit to target disfavored parties. This assumption is standard in regulatory frameworks and reflects institutional design rather than cryptographic guarantees. Mitigations for auditor misbehavior include: (1) *Multi-auditor verification*: High-stakes commitments can be verified by multiple independent auditors, analogous to second opinions in medical diagnosis; (2) *Auditor accountability*: Auditors who approve invalid proofs face professional sanctions, as with accounting fraud; (3) *Public verifiability*: Any party with access to the opened commitment can re-verify independently, making

auditor errors or fraud detectable. PCO scope is not regulatory capture or corrupt auditors; it does, however, provide an evidentiary trail to support accountability mechanisms.

Note on Input Authenticity. Input authenticity is explicitly out of scope: if a legal entity provides fraudulent inputs x (e.g., fabricated tax documents, falsified sensor readings), the proof π remains valid for those inputs. This is a feature, not a bug. PCO verifies *computation correctness* ("given inputs x, output y satisfies φ"), not *input provenance* ("inputs x are authentic"). Input authenticity requires orthogonal mechanisms: digital signatures from authoritative sources (e.g., employer-signed W-2 forms, sensor hardware attestations), notarization, or blockchain-based provenance tracking. The separation of concerns mirrors the design of cryptographic protocols: authentication and authorization are distinct properties. Specifications may require signed inputs (e.g., "W-2 must carry employer signature"), making input authenticity verifiable within the specification itself.

4.2 Security Definitions

We write λ for the security parameter and $\mathrm{negl}(\lambda)$ for any function negligible in λ. Let $\{0,1\}^\lambda$ denote the set of λ-bit strings. We write $\approx_c$ for computational indistinguishability.

Let $m := (q, \varphi, y, \pi, v, t_s)$ denote the message tuple consisting of query q, specification φ, output y, proof π, validator version v, and signed timestamp t_s. Define the commitment scheme as $\mathsf{Commit}(m;r) := \mathsf{H}(\mathsf{enc}(m,r))$, where enc is a canonical prefix-free encoding and $r \leftarrow \{0,1\}^\lambda$ is sampled uniformly.

Definition 3 (Binding)**.** The scheme is *binding* if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$,

$$\Pr\Big[(m,r,m',r') \leftarrow \mathcal{A}(1^\lambda) : m \neq m' \ \wedge\ \mathsf{Commit}(m;r) = \mathsf{Commit}(m';r')\Big] \leq \mathrm{negl}(\lambda).$$

Definition 4 (Hiding)**.** The scheme is *hiding* in the random oracle model if for all messages m_0, m_1:

$$\big\{\mathsf{Commit}(m_0;r) : r \leftarrow \{0,1\}^\lambda\big\} \approx_c \big\{\mathsf{Commit}(m_1;r) : r \leftarrow \{0,1\}^\lambda\big\}.$$

Definition 5 (Temporal Ordering)**.** Let $\mathcal{L}$ be an append-only ledger with timestamps. The scheme provides *temporal ordering* if: for any h appearing on $\mathcal{L}$ at time t_c with $h = \mathsf{Commit}(m;r)$, no PPT adversary can produce (m',r') satisfying $\mathsf{Commit}(m';r') = h$ and $m' \neq m$ after observing the ledger state at time t_c.

Note on Racing Attacks. Definition 5 considers an adversary who observes the ledger *after* commitment. A stronger adversary model would account for racing attacks, in which the adversary attempts to submit competing commitments simultaneously. Such attacks are mitigated by: (1) the binding property ensures the adversary cannot produce a different message with the same hash, (2) the nonce r is chosen (possibly globally and publicly broadcast) by the legal entity and kept secret until opening, so the adversary cannot predict h before commitment, and (3) blockchain consensus mechanisms (e.g., proof-of-work, proof-of-stake) provide total ordering of transactions, ensuring that even simultaneous submissions receive distinct timestamps. The delay requirement $t_e > t_c + \delta$ provides additional protection by ensuring the commitment is finalized before action.

Note on Validator Binding. The validator version v is included in the commitment tuple m, so validator binding follows directly from general binding (Definition 3).

4.3 Security Analysis

Theorem 1 (Binding)**.** *If* H *is collision-resistant, the scheme is binding.*

Proof. Suppose adversary $\mathcal{A}$ outputs (m,r,m',r') with $m \neq m'$ and $\mathsf{Commit}(m;r) = \mathsf{Commit}(m';r')$. Then $\mathsf{H}(\mathsf{enc}(m,r)) = \mathsf{H}(\mathsf{enc}(m',r'))$. Since enc is injective and $m \neq m'$, we have $\mathsf{enc}(m,r) \neq \mathsf{enc}(m',r')$. This constitutes a collision for H, contradicting collision resistance. □

Theorem 2 (Hiding)**.** *If* H *is modeled as a random oracle, the scheme is hiding.*

Proof. In the random oracle model, for any fixed m, the output $\mathsf{H}(\mathsf{enc}(m,r))$ is uniformly distributed over the range of H when r is uniform. Since the distribution depends only on the randomness r and not on m, commitments to distinct messages are identically distributed. □

Theorem 3 (Temporal Ordering)**.** *If* H *is collision-resistant and* $\mathcal{L}$ *is append-only with trusted timestamps, the scheme provides temporal ordering.*

Proof. Fix $h = \mathsf{Commit}(m;r)$ recorded on $\mathcal{L}$ at time t_c. Any adversary producing (m',r') with $\mathsf{Commit}(m';r') = h$ and $m' \neq m$ breaks binding. The append-only property ensures h cannot be removed or its timestamp altered after t_c. □

Corollary 1 (Audit Correctness)**.** *Let* $\mathsf{Verify}(\varphi,x,y,\pi)$ *be a sound verifier for specification* φ*. If the scheme is binding, then for any commitment* h *on* $\mathcal{L}$*: if* $\mathsf{Audit}(h,q,\varphi,y,\pi,r,v,t_s) = 1$ *and* $\mathsf{Verify}(\varphi,x,y,\pi) = 1$*, then output* y *satisfies specification* φ *for the committed query* q *and context* x *using validator version* v *at time* t_s*.*

Corollary 2 (Pre-Action Validation)**.** *The signed timestamp* t_s *in the commitment ensures that validation occurred before*

the action. Since t_s is bound in the commitment and issued by a trusted time service only after validator approval, the legal entity cannot act on the output before validation without creating a detectable inconsistency in the audit trail.

Note on Timestamp Service Dependency. Corollary 2 relies on a trusted time service. If this service is compromised or unavailable: (1) *Compromise*: A malicious time service could backdate timestamps, but this requires collusion with the legal entity and leaves evidence (the time service's signing key is implicated). Using multiple independent time services (e.g., from different jurisdictions) mitigates single-point-of-failure risks. (2) *Unavailability*: If the time service is unavailable, the protocol enters strict mode—no commitments can be made, and outputs are withheld. This design prioritizes auditability over availability. Production deployments should use established timestamp authorities (e.g., RFC 3161 compliant services) with redundancy and service-level agreements appropriate to the application's criticality.

5 Anticipations

5.1 RAG-Enhanced Proof Generation

General-purpose LLMs face systematic challenges when generating formal proofs: unfamiliar syntax, domain-specific type systems, and hallucinated library functions [53, 54]. However, compliance proofs operate in a highly constrained space where these challenges become tractable through RAG [29]. We propose that regulatory authorities publish *RAG knowledge bases* alongside formal specifications, enabling reliable proof generation through structured domain knowledge.

The regulatory authority that publishes a formal specification also publishes three artifacts: (1) Canonical Training Corpus, (2) Specification-Coupled SLM, (3) Certification Benchmark.

Authority-Published RAG Infrastructure. The regulatory authority that publishes a formal specification (e.g., `TaxCode.v`, `TrafficLaw.v`) also publishes three RAG artifacts:

(1) Canonical Example Corpus: Complete, verified proofs for representative scenarios covering normal operations, boundary conditions, and edge cases. These examples serve as few-shot learning templates, demonstrating correct syntax, type annotations, and proof patterns. Because the authority authored the specification, it understands the valid patterns and can provide authoritative examples.

(2) Syntax and Constraint Rules: Explicit documentation of domain-specific syntax requirements, including: proper import statements (`From Stdlib Require Import`), boolean vs. propositional comparison operators (`<?` vs. `<`), record field access syntax, and enumerated forbidden constructs that LLMs commonly hallucinate. These rules are machine-readable, enabling automatic prompt augmentation.

(3) Error-to-Fix Mappings: Comprehensive catalogs mapping common compilation errors to targeted corrections. For each error class (type mismatch, undefined reference, syntax error), the catalog provides a diagnostic pattern, an explanation, and a corrected code template. This enables systematic recovery from predictable LLM deviations.

This approach is analogous to how standards bodies publish reference implementations alongside specifications. The authority is best positioned to create training data RAG knowledge because it understands the edge cases, proof patterns, and common scenario pitfalls. Manufacturers can either use the authority-provided SLM directly, develop proprietary implementations using the authority's RAG knowledge base directly, or develop custom knowledge bases that pass the authority's certification benchmark validation suite.

Why RAG Achieves Reliable Proof Generation. The constrained proof space enables systematic reliability through RAG:

Challenge	RAG Solution	Mechanism
Unfamiliar syntax	Canonical examples	Few-shot learning
Hallucinated functions	Forbidden construct lists	Explicit constraints
Type mismatches	Error-specific hints	Targeted retrieval
Proof incompleteness	Pattern-guided regeneration	Structure transfer

The key insight is that formal proof compilation provides binary feedback: either the proof compiles or it does not, with specific error messages in the latter case. This deterministic feedback loop enables systematic error recovery through RAG. Unlike open-ended text generation, where quality is subjective, proof generation has an objective oracle (the type checker), making RAG-based recovery tractable and predictable.

Critically, regulatory authorities are responsible for validating that their published formal specifications—and accompanying RAG knowledge bases—are compatible with current LLM proof generation capabilities *before* requiring legal entities to comply with them. This pre-deployment validation ensures that RAG-enhanced proof generation achieves reliable performance. The smart contract industry demonstrates this principle in practice: Ethereum specifications are designed with existing toolchains in mind, and complex specifications are decomposed into tractable sub-proofs.

Deployment Modes. For deployments using authority-published RAG knowledge bases, or for novel scenarios where RAG refinement is ongoing, we identify three operational modes:

Mode 1: Standard RAG Pipeline. The full three-stage pipeline (initial generation, error-guided refinement, pattern-guided regeneration) handles proof generation automatically.

Appropriate for the majority of compliance scenarios where the RAG knowledge base covers the required patterns.

Mode 2: Ensemble RAG. Multiple LLMs with RAG augmentation generate proofs in parallel, with the first verified proof accepted. Appropriate for latency-sensitive applications or scenarios requiring additional redundancy.

Mode 3: Supervised RAG. For novel scenarios outside the RAG knowledge base, human experts review and extend the knowledge base. Appropriate during initial deployment or when regulations change significantly.

Our case studies demonstrate that RAG-enhanced LLMs generate reliable proofs across a range of complexities: basic tax arithmetic, multi-criteria product recommendations, and autonomous vehicle safety constraints. Within the predicate scope defined in Section 1, RAG plus authority-published examples yields measured success rates of 78.7% (Coq, large models) to 95% (STL, SLMs); we discuss residual failure modes in Section 5.1.

5.2 Specification Authorship and Governance

PCO verifies compliance with formally stated specifications, but does not guarantee that these specifications themselves faithfully capture the intended policy, regulation, or normative goal. An underspecified or poorly designed specification may admit undesirable behaviors while remaining formally compliant. We identify specification governance as a critical prerequisite for PCO deployment.

We acknowledge that formalizing law is genuinely difficult and remains an open research area. Statutory and especially case-law text is intentionally open-textured, and a fully faithful formalization of an entire legal regime is not currently feasible. We therefore make no claim that arbitrary regulations can be captured in φ. What PCO can do today is verify compliance with the *stated, machine-checkable predicates* a regulator chooses to publish, leaving the legislative-to-formal translation as a separate (and substantial) workflow handled by tools such as Catala [33] and Symboleo [47] together with domain experts.

Specification authorship typically falls to government agencies, as illustrated by our tax and autonomous vehicle case studies.

Authorship Challenge. Tax codes are written by legislators and legal experts, not formal methods specialists. Production deployment requires formal methods experts or validated translation tools. While smart contracts and projects such as Catala demonstrate that *some* legal rules can be expressed algorithmically, this falls far short of arguing that legal specification is tractable in general. Statutory ambiguity, contextual interpretation, and case-law evolution remain hard problems we do not solve.

Validation Paradox. If humans struggle to verify LLM outputs, how can they verify specifications? Practical solutions exist: (1) automated testing against known valid/invalid inputs, (2) formal consistency checking, (3) bug bounty programs, and (4) incremental deployment. The key insight: *verifying specification behavior on test cases* is tractable even when *understanding the specification abstractly* is not. We propose multi-tiered validation: domain expert review, formal methods review, scenario-based testing [12, 13], and public review periods. See 6 for details.

Evolution and Maintenance. Specifications must be versioned; the commitment includes version φ. Legacy commitments remain valid under old specifications. Scenario-driven versioning creates a virtuous cycle: deployment surfaces edge cases, which improve specifications, which train better SLMs. Scenario-driven versioning, backward compatibility, and complexity estimates are in 6

5.3 Validator Ecosystem and Reputation

The PCO protocol assumes legal entities employ "reputable validators" to verify proofs before commitment, but provides no mechanism for establishing validator reputation. We outline a validator ecosystem design, drawing on analogous mechanisms in existing trust infrastructures.

Validator Certification Framework. We propose a three-tier model: (1) Self-Assessment for low-stakes applications, (2) Third-Party Assessment by certified validators (analogous to PCI-DSS QSAs), and (3) Regulatory Validators for highest-stakes applications. The commitment records validator tier and certification identifier.

Collusion Resistance. Validators execute deterministic algorithms on open-source proof checkers, reducing trust to software integrity rather than human trustworthiness. Rubber-stamping is addressed through verification evidence requirements and random meta-validator audits.

Legal Integration and Incentives. Validator attestations carry legal weight through contractual liability, professional licensure (analogous to CPAs), and regulatory safe harbors. Economic incentives—fees, reputation, and bonding—create honest verification incentives. Failed validators face blacklisting. Full details in 7

Concrete Anti-Rubber-Stamping Mechanisms. A natural question is what concrete mechanisms would prevent the emergence of validators who approve proofs without meaningful verification. We propose three complementary mechanisms, drawing on PKI, financial auditing, and proof-of-stake protocols:

1. *Economic bonding.* Validators post a financial bond as a precondition for accreditation. Any approval of a proof later shown by re-verification to be invalid triggers proportional slashing. This converts rubber-stamping from a free action into an actuarially priced risk—the same logic that underpins proof-of-stake consensus and surety bonds in commercial law.
2. *Randomized cross-validation.* A fraction ρ of committed proofs (e.g., ρ=0.01 in routine deployment, ρ=1.0 in high-stakes domains) is independently re-verified by a different validator drawn from a separate pool. Persistent rubber-stamping by a validator becomes statistically detectable: if validator v_1 approves proofs that fail ρ-sampled re-verification at rate exceeding a published threshold, v_1 is flagged. This is the same mechanism CT logs use to detect misbehaving CAs.
3. *Public attribution.* Every committed tuple carries the validator's version identifier v in the hash preimage. Validator behavior is therefore publicly attributable across time. Combined with bonding and cross-validation, this enables both ex-ante deterrence and ex-post forensics. Researchers, journalists, and competing validators all have standing to challenge a validator's track record.

This is not a trust-by-assumption model: it is the same accountability stack that already governs certificate authorities under the WebPKI [16] and that financial regulators apply to external auditors [11]. We therefore reject the framing that PCO "shifts rather than solves" the trust problem: it converts trust in opaque AI behavior into trust in publicly auditable validators backed by economic and legal incentives.

The preceding subsections address deployment considerations. Section 5.4 provides empirical evaluation of proof generation performance. PCO requires specifications with unambiguous formal semantics and an append-only ledger with trusted timestamps.

5.4 Experimental Evaluation

We conducted a systematic performance evaluation of our PCO framework implementation across multiple large language models and complexity levels.

5.4.1 Experimental Setup

LLM Providers: We evaluated five state-of-the-art LLMs representing diverse architectural approaches: (1) OpenAI GPT-4 Turbo (commercial, closed-source), (2) Anthropic Claude 3.5 Sonnet (commercial, safety-focused), (3) Google Gemini 2.0 Flash (commercial, optimized for speed), (4) Meta Llama 3.3 70B via Groq (open-source, hardware-accelerated), and (5) DeepSeek Chat (open-source, reasoning-focused)

Use Cases and Complexity Levels: We tested two representative compliance scenarios, each at three difficulty levels: For tax compliance: (1) *Easy*—single W-2, standard deduction, (2) *Medium*—multiple income sources, itemized deductions, child credit, and (3) *Hard*—self-employment, capital gains, QBI deduction, multi-state.

For recommendation systems: (1) *Easy*—2 laptops with clear winner, (2) *Medium*—10 laptops with weighted multi-criteria utility, and (3) *Hard*—100 laptops with transparency disclosure requirements

Methodology: For each (LLM, use case, complexity) tuple, we executed 5 independent iterations, measuring: (1) proof generation time (LLM API call), (2) Coq verification time, (3) generated proof size in lines of code, and (4) token consumption (input and output). Total: 150 test executions (5 models $\times$ 2 use cases $\times$ 3 complexity levels $\times$ 5 iterations).

Statistical Considerations: We chose 5 iterations per configuration to balance experimental coverage (150 total executions across 30 configurations) with API costs and time constraints. For RAG stage distribution analysis, 5 iterations per configuration provide meaningful signal about which RAG stages are typically required for each (model, complexity) combination. For timing measurements, we report means; variance was low (coefficient of variation $< 15\%$) as LLM generation time is dominated by deterministic factors (prompt length, output length) rather than stochastic variation.[3]

Complexity Level Criteria: Difficulty levels are defined by formal criteria measuring specification complexity: (1) *Proof obligations*: Easy scenarios require 1–3 sub-goals, medium 4–8, hard 10+. (2) *State space size*: Easy involves < 10 entities (forms, products), medium 10–50, hard 50+. (3) *Logical depth*: Easy uses propositional logic with arithmetic, medium adds bounded quantification, hard requires nested quantifiers and recursive definitions. (4) *Type complexity*: Easy uses primitive types, medium introduces records and lists, hard requires dependent types or complex inductive structures. These criteria ensure reproducibility and enable systematic study of LLM capabilities across the complexity spectrum.

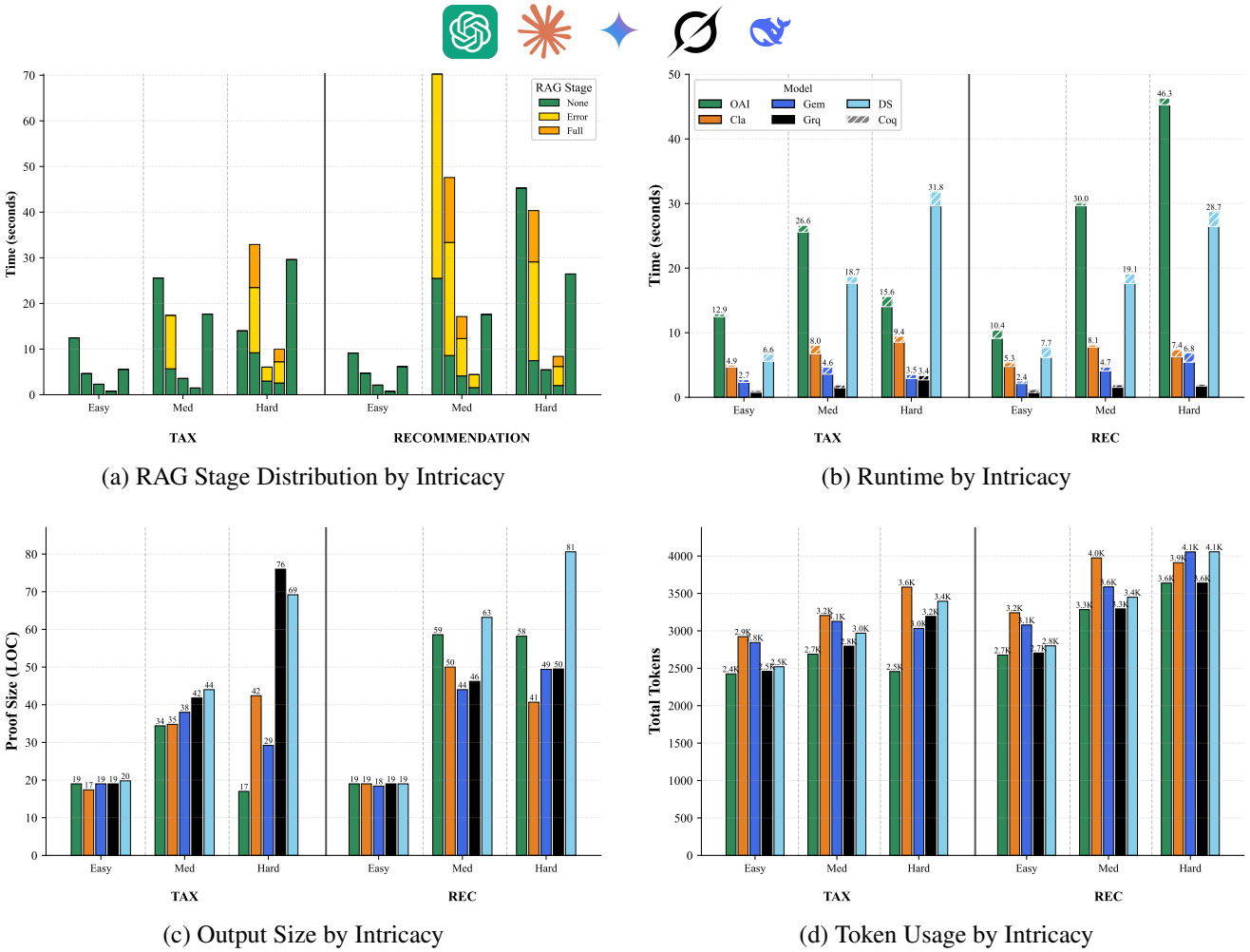
5.4.2 Results

Figure 2 presents comprehensive performance metrics across all experimental conditions.

RAG-Enhanced Proof Generation (Figure 2a): Our RAG pipeline augments LLM prompts with domain-specific knowledge to ensure reliable proof generation. The pipeline operates in three stages:

- *Stage 1 (Initial Generation)*: The LLM receives the specification along with complete working examples from the target domain (tax, recommendation, or autonomous vehicle). These examples demonstrate correct Coq syntax,

[3] Coefficient of variation (CV) per model: GPT-4 Turbo: 11.2%, Claude Sonnet: 9.8%, Gemini Flash: 13.4%, Groq/Llama: 8.1%, DeepSeek: 12.7%. CV computed as σ/μ across iterations within each (model, scenario) pair.



(a) RAG Stage Distribution by Intricacy

(b) Runtime by Intricacy

(c) Output Size by Intricacy

(d) Token Usage by Intricacy

Figure 2: PCO framework performance across 5 LLMs, 2 use cases (Tax, Recommendation), and 3 complexity levels (Easy, Medium, Hard). All metrics averaged over 5 iterations per configuration. Runtime graph (b) shows LLM generation time (solid) and Coq verification time (white diagonal hatching). Token usage graph (d) shows input tokens (solid) and output tokens (white diagonal hatching).

proper import statements, type annotations, and proof patterns. Critical syntax rules are explicitly stated, including forbidden constructs that LLMs commonly hallucinate (e.g., `option_get`, `len`).

- *Stage 2 (Error-Guided Refinement)*: If the initial proof does not compile, the error message is analyzed, and domain-specific hints are retrieved. For example, type mismatch errors trigger hints about Coq's numeric type system (`nat` vs `Z`), while undefined reference errors provide correct alternatives for commonly hallucinated functions.

- *Stage 3 (Pattern-Guided Regeneration)*: For complex specifications, the LLM receives the complete working pattern with explicit instructions to adapt the structure while preserving the logic. Post-processing automatically corrects systematic issues: converting comparison operators to boolean form (`<` to `<?`), ensuring proper imports, and standardizing proof termination.

RAG Knowledge Sources: The RAG system draws from three authoritative knowledge sources that regulatory authorities can publish alongside specifications: (1) *Canonical examples*: Complete, verified Coq code for representative scenarios in each domain, serving as few-shot learning templates; (2) *Syntax rules*: Critical Coq syntax requirements, including the modern `From Stdlib Require Import` syntax and boolean comparison operators; (3) *Error catalogs*: Mappings from common error patterns to targeted corrections, enabling the system to recover from predictable LLM mistakes.

Proof Generation Time (Figure 2b): Runtime increased with intricacy. For easy scenarios: Groq achieved sub-1s generation (0.93s average), followed by Gemini (2.8s), Claude

(3.9s), OpenAI (5.9s), and DeepSeek (7.1s). For hard scenarios, times ranged from 2.5s (Groq) to 28s (DeepSeek). Coq verification remained consistently fast (<0.5s) across all successful proofs, visible as thin white-hatched layers atop the solid LLM time bars. This confirms that formal verification overhead is negligible compared to proof generation.

Output Size (Figure 2c): Proof sizes scaled with scenario intricacy. Easy scenarios: 30–50 lines; medium: 50–80 lines; hard: 80–150 lines. Tax scenarios generally produced more concise proofs than recommendation scenarios due to arithmetic simplicity versus multi-criteria optimization logic. Model-specific variations reflect different proof styles: Gemini generated the most compact proofs, while DeepSeek produced more verbose intermediate steps.

Token Usage (Figure 2d): Total token consumption ranged from 600 tokens (easy scenarios) to 1,400 tokens (hard scenarios). Input tokens (system prompt + specification, shown as solid bars) increased modestly with intricacy (400–700 tokens), while output tokens (generated proofs, shown as white-hatched overlay) scaled more dramatically (200–700 tokens). At commercial API rates ($0.01–0.05 per 1K tokens), per-proof costs ranged from $0.006 (easy) to $0.07 (hard).

5.4.3 Discussion

Our experiments demonstrate both the promise and current limitations of LLM-based proof generation for PCO:

Key Finding: RAG-enhanced proof generation achieves reliable performance across all complexity levels. The three-stage pipeline—initial generation with examples, error-guided refinement, and pattern-guided regeneration—addresses the systematic challenges LLMs face with formal language syntax. This approach validates that PCO deployments can achieve dependable proof generation by combining LLM capabilities with structured domain knowledge.

RAG Effectiveness by Complexity: Figure 2a shows the distribution of RAG stages required across complexity levels. Easy scenarios are predominantly completed in the initial stage, demonstrating that well-crafted examples enable LLMs to generate correct proofs directly. In medium- and hard-scenario settings, error-guided refinement and pattern-guided regeneration increasingly benefit, with the RAG system recovering from initial syntax or type issues through targeted knowledge retrieval.

Why RAG Works for Formal Proofs: Unlike open-ended text generation, formal proof generation has a precisely defined correctness criterion: the proof must compile. This binary feedback enables systematic error recovery. When a proof fails to compile, the error message provides specific, actionable information (line number, expected vs. actual type, undefined reference). The RAG system maps these error patterns to domain-specific corrections, transforming unpredictable LLM behavior into bounded, recoverable deviations.

Failure Modes Where RAG Does Not Help. It is important to be explicit about cases where proof generation systematically fails even with RAG enhancement, so that the capability boundary is visible. The recurring failure classes we observed across 78.7% Coq success and 95% STL success are:

- *Non-decidable subgoals.* When a synthesized subgoal lies outside Rocq's automation (e.g., requires inductive hypotheses that the LLM does not name), no amount of error-to-fix retrieval recovers it; only pattern-guided regeneration with a matching template helps.
- *Cross-clause interactions.* Specifications composed of ≥ 5 conjoined predicates with shared variables (e.g., QBI deduction with phase-out) defeat templates that handle each clause in isolation. Failures are silent: the LLM emits a plausible but unsound proof that the checker rejects.
- *Out-of-distribution outputs.* If the LLM produces y whose structure is not anticipated by any template in the corpus, RAG cannot supply a relevant example. The system then either falls through to human review or fails.

These failure modes mark the boundary of current capability. They are not arguments against PCO as a security architecture: an unverifiable output is simply not committed, and no audit trail is generated for it. They are arguments for richer authority-published corpora and for human-in-the-loop fallback (*Mode 3* in Section 5.1).

Model-Specific Trade-offs: Groq's hardware acceleration delivers exceptional speed (3–5× faster than competitors). Claude excels at following structured examples. Gemini balances speed and precision. DeepSeek produces thorough intermediate reasoning. These complementary strengths suggest that PCO deployments may benefit from ensemble approaches: attempting multiple models in parallel and accepting the first verified proof.

Practical Deployment Strategy: We recommend RAG-enhanced deployment with optional ensemble redundancy:

- *Tier 1 (Simple specifications)*: Single model with RAG augmentation; initial generation typically sufficient
- *Tier 2 (Medium complexity)*: Single model with full RAG pipeline (all three stages); ensemble approach for latency-sensitive applications
- *Tier 3 (High complexity)*: Ensemble mode with 2–3 models using RAG pipelines in parallel, accepting first verified proof

Preferred Deployment: Authority-Published RAG Knowledge Bases. The recommended deployment uses specification-coupled RAG knowledge bases published by regulatory authorities alongside formal specifications (Section 5.1). These knowledge bases contain: (1) canonical

working examples for all specification categories, (2) domain-specific syntax rules and constraints, and (3) comprehensive error-to-fix mappings validated against the specification's type system. Legal entities either use the authority's RAG knowledge base directly or develop custom knowledge bases that pass the authority's certification benchmark.

Ensemble and RAG Synergy Different LLMs exhibit complementary strengths: GPT-4 achieves higher precision on tax scenarios requiring arithmetic reasoning, while Claude Sonnet excels at following structured examples. The RAG pipeline normalizes model-specific variations by providing consistent domain knowledge, whereas ensemble approaches leverage model diversity to optimize latency. A practical deployment attempts to generate proofs in parallel using multiple RAG-enhanced models, accepting the first verified proof.

Cost and Scalability: At $0.006–0.07 per proof and 3–28 seconds per generation, PCO is economically viable for high-stakes decisions. Generating 1,000 daily compliance proofs costs $6–70/month in API fees and completes within minutes using parallel API calls—negligible compared to regulatory penalties or litigation costs. Storage requirements (30–150 lines × 5 iterations = 30KB per scenario) are modest; off-chain proof storage with on-chain Merkle commitments scales to billions of proofs.

5.4.4 Hard Real-time PCO with Temporal Logic

While static Coq proofs suffice for policy-driven systems, safety-critical applications require real-time verification. We evaluated an alternative PCO architecture using SLMs to generate STL specifications verified by RTAMT [39].

System Architecture: SLMs generate STL specifications encoding temporal safety properties. For example, a turn maneuver specification: `always[0:100](speed <= 32.0 and lateral_accel <= 3.5)` requires speed below 32.0 m/s and lateral acceleration below 3.5 m/s^2 throughout a 100-timestep window. RTAMT verifies these against execution traces using offline monitoring, producing quantitative robustness degrees ρ ($\rho > 0$ indicates satisfaction).

Experimental Setup: Four SLMs—Llama 3.2 3B, Phi-3 Mini 3.8B, Qwen 2.5 7B, CodeLlama 7B—evaluated across 80 tests (2 maneuvers × 2 intricacies × 4 models × 5 iterations).

Results: (Figure 3) 151ms average latency (122ms generation + 29ms verification), approaching 100ms target. Hash computation adds <1ms; ledger commitment 1–15s. The SLM-STL approach achieves 20–185× speedup over Coq proofs with ∼100× cost reduction ($0.0002 vs $0.006–0.07).

Proof Template Libraries: Pre-computed parameterized proof skeletons for common maneuvers (lane changes, turns, braking) provide bounded worst-case latency through parameter substitution rather than full generation. Novel situations fall back to full generation.

Limitations: Template-based verification is sophisticated

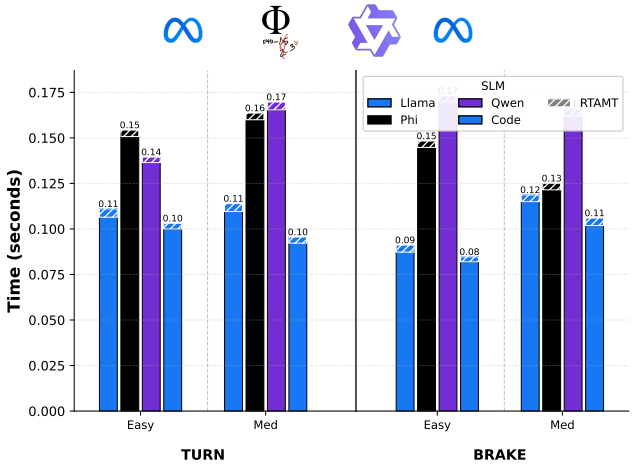


Figure 3: Real-time PCO verification latency for autonomous vehicle maneuvers. Bars show SLM generation time (solid) stacked with RTAMT verification time (white diagonal hatching). Average total latency of 151ms closely approaches the 100ms (10Hz) decision cycle required for autonomous vehicles. Results shown for turn and brake maneuvers at easy and medium intricacy levels. Four SLMs tested: Llama 3.2 3B (dark blue), Phi-3 Mini (black), Qwen 2.5 7B (violet), CodeLlama 7B (light blue).

table lookup—useful for engineering robustness but not demonstrating general proof generation. The 151ms latency for full STL generation better represents current capabilities.

Deployment Strategy: Hybrid two-layer architecture: (1) Offline regulatory layer with large models for Coq proofs (3–28s), (2) Real-time operational layer with SLMs for STL specifications (151ms), (3) Unified commitment binding both layers.

5.5 On Superintelligence

As superintelligence becomes an anticipated rather than speculative development [22], accountability infrastructure must scale accordingly. PCO provides a foundation that does not depend on understanding AI reasoning—only on verifying outputs against specifications. Whether AI systems exhibit emergent capabilities or unexpected behaviors [6, 51], PCO ensures that whatever they produce must pass through a formal verification gate before action. This positions PCO as critical infrastructure for the superintelligence era: not a solution to alignment, but a necessary layer of accountability that makes compliance auditable regardless of how advanced AI systems become.

6 Specification Governance Details

Multi-Tiered Validation Process:

(1) Domain Expert Review: Legal/regulatory experts validate that the formal specification captures regulatory intent, even if they cannot read Coq directly.
(2) Formal Methods Review: Proof engineers verify that the specification correctly encodes the domain expert's requirements.
(3) Test-Based Validation: Specification tested against known compliant and non-compliant examples. Includes scenario-based testing inspired by Live Sequence Charts, where desired scenarios are verified to be encapsulated while undesired scenarios are excluded. The scenario corpus validates specification correctness and trains specification-coupled SLMs.
(4) Public Review Period: Draft specifications published for stakeholder comment before adoption.

Scenario-Driven Versioning: When real-world cases reveal specification gaps, the authority adds scenarios to the test corpus, publishes corrected versions, and retrains SLMs. This creates a virtuous cycle: deployment surfaces edge cases $\rightarrow$ edge cases improve specifications $\rightarrow$ improved specifications train better SLMs.

Backward Compatibility: (scenario, proof) pairs valid under version n remain valid under version $n+1$ unless representing known bugs.

Specification Complexity Estimate: IRS Section 199A (QBI deduction) estimated at 430–630 lines: Core QBI (50–80), Phase-out (100–150), SSTB determination (80–120), Aggregation elections (150–200), REIT/PTP provisions (50–80).

Democratic Specification: For domains without clear regulatory authorities, specification standards bodies (analogous to W3C) and blockchain-based governance mechanisms may enable decentralized specification approval.

7 Validator Ecosystem Details

Three-Tier Certification Framework:
Tier 3 (Self-Assessment): Organizations run standard open-source proof checkers internally. Legal entity assumes full liability.
Tier 2 (Third-Party Assessment): Independent validators certified by domain-specific regulatory bodies. Certification requires: (a) demonstrated competence through test suites, (b) liability insurance/bonding, (c) annual audits, (d) secure SDLC documentation.
Tier 1 (Regulatory Validators): Government-operated validators for highest-stakes applications with full regulatory backing.

Validator Market Structure: Validators compete in a reputation-based market. Selection criteria: track record, independence, audit transparency, liability acceptance (bonded validators stake assets).

Legal Framework Integration: Three pathways: (1) Contractual liability through service agreements, (2) Professional licensure analogous to CPAs/PEs, (3) Regulatory safe harbor provisions shifting liability to validators.

Multi-Validator Requirements: High-stakes decisions may require multiple independent validators. Commitment records validator list (v becomes a list).

Economic Incentives: Validators face reputation damage, financial penalties (bond forfeiture), and regulatory sanctions (license revocation) for approving invalid proofs.

Fraud Detection: Invalid approvals detected during audit when $\mathsf{Verify}(\varphi, x, y, \pi) = 0$ despite approval. Committed v identifies fraudulent validator; repeated failures lead to blacklisting.

Specification Security: Open-source specifications undergo continuous stakeholder review. Flaws are corrected in versioned updates; commitment includes version φ so legacy commitments remain valid.

Adversarial Inputs: User inputs carry digital signatures from originating authorities. Legal entity bears responsibility for fraudulent inputs—proof certifies compliance *given inputs*, not input authenticity.

8 Concluding Remarks

Existing approaches to trustworthy AI fall into three disjoint categories: explanations that attempt to persuade humans, verifiable computation that proves correct execution of a fixed program, and audit logging that records what happened without certifying correctness. Proof-Carrying Output occupies a previously unaddressed point in this design space. Unlike explainable AI, PCO provides no post-hoc rationalization and does not rely on human judgment; unlike SNARKs and verifiable computation, PCO does not prove execution of a predetermined circuit but certifies compliance of novel, dynamically generated outputs against expressive, evolving specifications; and unlike signed logs, PCO cryptographically binds correctness evidence to the output before action, preventing retroactive justification. The key shift is architectural rather than incremental: trust is moved from opaque model behavior and unverifiable explanations to a small, auditable proof checker whose correctness is independent of the AI system. This "prove-then-act" paradigm generalizes Proof-Carrying Code from static software to generative AI decisions, establishing a new accountability primitive for high-stakes AI deployment that existing mechanisms cannot subsume.

PCO requires that specifications have unambiguous formal semantics. Natural-language user queries must therefore be mapped into well-defined formal predicates through domain-specific interfaces. Ambiguous or underspecified queries cannot be meaningfully verified. Accordingly, we focus on structured query formats (e.g., parameterized or multiple-choice queries) with precise formal interpretations, rather than attempting to directly formalize arbitrary free-form natural language.

The protocol relies on an append-only ledger providing trusted timestamps. Commitments bind outputs, specifications, and proofs to a verifiable execution context. If the ledger is unavailable, commitment is prevented, and outputs are withheld in strict mode. This design choice prioritizes auditability and non-repudiation over availability.

PCO extends PCC from static programs to dynamic AI outputs. PCO transfers trust from LLMs to verifiable certificates via machine-checkable proofs and cryptographic commitments. The query-proof binding mechanism prevents proofs from being reused for different queries or specifications after the commitment is made. Our implementations demonstrate feasibility across three domains: tax compliance, product recommendations, and real-time operational safety monitoring (151ms latency using STL/RTAMT). Critically, our experiments demonstrate that RAG-enhanced proof generation achieves reliable performance by combining LLM capabilities with structured domain knowledge—canonical examples, syntax rules, and error-to-fix mappings published by regulatory authorities. The shared infrastructure demonstrates PCO's scalability across the latency-expressiveness spectrum: from complex offline reasoning with large models to near-real-time monitoring with small models, both producing verifiable commitments to immutable ledgers.

For the protocol to be practical, three elements must be in place: (i) publication of formal specifications by domain authorities with robust governance (Section 5.2), (ii) establishment of a validator ecosystem with reputation mechanisms (Section 5.2), and (iii) deployment of append-only ledgers that have regulatory recognition. As these elements develop, PCO can establish a foundation for accountable AI.

There exist several variants of the framework, one possibility is that proof verifiers are responsible to record on the blockchain their commitment on validity of the proof, and the user is notified by a smart contract that a proof for the action has been verified and committed before the user takes action.

9 Case Study I: Tax Computation

9.1 Problem

Tax codes are complex (exhibit combinatorial intricacy) due to interactions among deductions, credits, and phase-outs. Poses difficulties for optimization, making it hard for individuals to navigate effectively and limiting what humans often struggle to overcome. While an LLM may propose a reasonable strategy, it might fail to consider optimal alternatives. Moreover, users typically lack the expertise necessary to validate claims of optimality.

9.2 Specification

We specify the correctness of a tax return claim as a logical predicate in Rocq. The predicate states that all values declared in the claim must coincide with values recomputed from the authoritative inputs: the supporting tax documents and the filing status. This specification was developed by the authors to demonstrate the PCO framework; production deployment would require IRS-authored and validated specifications (see Section 5.2).

```
1 (* From ComputeSpec.v *)
2
3 Definition claim_is_correct
4   (docs   : TaxDocuments)
5   (status : filing_status)
6   (claim  : TaxReturnClaim) : Prop :=
7   let agi := compute_agi_simple docs in
8   let taxable := compute_taxable_income_std
9                    agi status in
10  let tax := compute_tax_by_status
11               taxable status in
12  let withheld := compute_total_withheld docs in
13    claim_agi claim = agi /\
14    claim_taxable_income claim = taxable /\
15    claim_tax_liability claim = tax /\
16    claim_total_withheld claim = withheld /\
17    claim_refund claim = withheld - tax.
```

Explanation. Line 3 introduces `claim_is_correct` as a definition returning a proposition (`Prop`), indicating that correctness is expressed as a logical property to be proven, rather than as a Boolean decision procedure.

Lines 4–6 define the inputs of the specification: `docs` represents the authoritative tax documents, `status` represents the taxpayer's filing status, and `claim` represents the submitted tax return whose correctness is being verified.

Line 7 recomputes the *adjusted gross income* (AGI) directly from the source documents. This establishes AGI as a derived quantity whose value does not depend on the claim itself.

Line 8 computes the taxable income from the recomputed AGI and the filing status, reflecting the application of the standard deduction rules associated with that status.

Line 10 derives the tax liability as a function of the taxable income and filing status. This separates the tax computation logic from any values asserted in the claim.

Line 12 aggregates the total amount of tax withheld at source, again using only the supporting documents.

Lines 13–16 assert that each corresponding field in the submitted claim exactly matches the recomputed values for AGI, taxable income, tax liability, and total withheld amount.

Finally, Line 17 enforces the algebraic definition of the refund as the difference between the total amount withheld and the computed tax liability. This definition is not recomputed independently, but derived from previously established quantities, making the dependency explicit.

Overall, the specification characterizes a correct tax claim as one whose declared values are consistent with a deterministic recomputation from trusted inputs, providing a clear separation between computation and verification.

Specification Satisfaction vs. Correctness. PCO guarantees that outputs satisfy published specifications; it does not guarantee that specifications capture all relevant requirements. A valid proof demonstrates specification satisfaction, not absolute correctness. If the specification is incomplete, a valid proof may provide limited assurance for aspects not covered. Specification completeness is a governance challenge addressed through public review, adversarial testing, and iterative refinement (Section 5.2). Specifications can also encode fundamental logical axioms—consistency, transitivity of preferences, arithmetic correctness—providing baseline guarantees independent of regulatory completeness.

The specification assumes that the provided tax documents are authentic, internally consistent, and correctly attributed to the taxpayer; it verifies the correctness of the claimed computation relative to those documents, not the absence of omitted, fraudulent, or undisclosed income.

9.3 Instantiation

Let x denote the taxpayer's certified documents, and let y denote the values proposed in the tax return. An LLM generates a proof π_{tax} such that $\mathsf{Verify}(\varphi_{\text{tax}}, x, y, \pi_{\text{tax}}) = 1$. Prior to filing, the system commits $h \leftarrow \mathsf{H}(q \,\|\, \varphi_{\text{tax}} \,\|\, y \,\|\, \pi_{\text{tax}} \,\|\, r)$ to the append-only ledger $\mathcal{L}$.

We illustrate this process with a concrete instantiation taken from our Rocq implementation. The example models a taxpayer with $75,000 in wages and $500 in interest income.

```
1 (* From TaxProof_Example.v *)
2
3 Definition jane_docs : TaxDocuments := mkTaxDocs
4   "***-**-1234" 2024
5   (jane_w2 :: nil)        (* W-2: \$75,000 wages *)
6   (jane_1099int :: nil)   (* 1099-INT: \$500 interest *)
7   nil nil nil.
8
9 Definition jane_claim : TaxReturnClaim := mkClaim
10   8451                                  (* Tax liability *)
11   12000                                 (* Total withheld
       *)
12   3549                                  (* Refund *)
13   75500                                 (* AGI *)
14   60900.                                (* Taxable income
       *)
15
16 Theorem jane_return_correct :
17   claim_is_correct jane_docs Single jane_claim.
18 Proof.
19   (* proof script elided; see artifact for full Rocq
       development *)
20 Qed.
```

Explanation. Line 3 defines `jane_docs`, an instance of `TaxDocuments`, constructed using the helper `mkTaxDocs`. This value represents the authoritative input x to the verification predicate.

Line 4 provides identifying metadata (redacted identifier and tax year), which is included in the document record but plays no role in the arithmetic of the tax computation.

Lines 5 and 6 supply the income sources: a single W-2 form reporting $75,000 in wages and a single 1099-INT form reporting $500 in interest income. Line 7 indicates that no other document categories (e.g., additional income, adjustments, or credits) are present.

Line 9 defines `jane_claim`, an instance of `TaxReturnClaim`, representing the proposed return values y. The fields correspond to the quantities constrained by the specification in Section 9.2.

Lines 10–14 enumerate the claimed numerical values: the tax liability, total withheld amount, resulting refund, AGI, and taxable income. These values are intended to match the results of the deterministic recomputation performed by `claim_is_correct`.

Line 16 states the main correctness theorem for this instance. Line 17 asserts that the concrete documents `jane_docs`, together with filing status `Single`, satisfy the specification when paired with the claimed return `jane_claim`.

The proof (elided here) establishes that all equalities required by `claim_is_correct` hold for these concrete values. The final `Qed` at Line 20 certifies the claim as formally verified, providing a machine-checkable witness π_{tax} that can be committed to the ledger as described above.

The system constrains only the correctness of the tax return claims: a taxpayer may submit an incorrect return, but cannot obtain a verifiable compliance certificate for any claim that deviates from the specification.

9.4 Audit

During an audit, the taxpayer submits the data $(q, y, \pi_{\text{tax}}, r)$ to the auditor, denoted as Verifier. The auditor verifies the integrity of the data by checking whether $\mathsf{H}(q \,\|\, \varphi_{\text{tax}} \,\|\, y \,\|\, \pi_{\text{tax}} \,\|\, r) = h$ and then runs the verification process, Verify. While the proof contains sensitive information, such as income sources or deduction strategies, committing only to the hash ensures that all details remain hidden until the audit.

Existing internal algorithmic auditing workflows [43] can incorporate the audit process. Rather than depending solely on documentation and log inspections, auditors can obtain a mechanically verifiable certificate confirming that the filed return complies with a published formal specification.

Note that if a taxpayer fails to include income from x, the proof will still be valid for the declared inputs; any fraudulent activity falls outside the scope of this formal model.

10 Case Study II: Autonomous Vehicles

10.1 Problem

When an autonomous vehicle (AV) performs a maneuver, it creates uncertainty. Questions arise from this situation: did the system malfunction? , or did it operate as intended? Was the resulting harm preventable, or was it an unavoidable consequence of physical laws?

Koopman and Wagner highlight that current validation and logging mechanisms do not suffice to explain failures of ML-based AV components or to rule out overfitting and rare black swan scenarios [27]. Standard event data recorders and telemetry logs capture what happened, what sensors and actuators did, but fall short of providing an understanding of the reasoning behind the system's decisions. This limitation makes it exceedingly difficult for investigators to discern between an internal failure (where the system received critical warning information but failed to act) and an external failure (where the system executed its function correctly, but an unavoidable physical outcome ensued).

10.2 Specification

We formalize vehicle compliance as logical predicates over the current vehicle state, the surrounding environment, and (where applicable) the intended driving action. The Rocq development specifies independent predicates for speed-limit compliance, safe following distance, and traffic signal compliance.

```
1 (* From ComplianceSpec.v *)
2
3 Definition speed_limit_compliant
4   (v : vehicle_state) (env : environment)
5   (act : driving_action) : Prop :=
6   let projected := veh_speed v
7                    + act_accel act * 1000 in
8   let limit := seg_speed_limit
9                  (env_map env) in
10  projected <= limit.
11
12 Definition following_distance_compliant
13   (v : vehicle_state) (env : environment) : Prop :=
14   forall obj, In obj (env_objects env) ->
15     obj_class obj = ObjVehicle ->
16     let gap := distance_to_object v obj in
17     let required := required_following_distance
18                       v env in
19     gap >= required.
20
21 Definition signal_compliant
22   (v : vehicle_state) (env : environment) : Prop :=
23   forall sig, In sig (env_signals env) ->
24     signal_requires_stop sig ->
25     veh_speed v = 0 \/ veh_accel v < 0.
```

Explanation. Line 3 defines `speed_limit_compliant` as a predicate returning a proposition, indicating that speed compliance is treated as a logical property rather than a Boolean check.

Lines 4–5 specify the inputs to the predicate: the current vehicle state `v`, the surrounding environment `env`, and a proposed driving action `act`. This reflects that speed compliance depends not only on the current speed but also on the intended acceleration.

Line 6 computes the projected vehicle speed by extrapolating the current speed with the acceleration implied by the action. The constant factor captures the chosen time horizon for projection and is fixed by the model.

Line 8 retrieves the applicable speed limit from the map segment associated with the environment. Line 10 enforces compliance by requiring the projected speed to remain below the legal limit.

Line 12 introduces `following_distance_compliant`, a predicate that expresses safe-following behavior independently of any specific action.

Lines 13–14 universally quantify over all objects present in the environment. Line 15 restricts the condition to objects classified as vehicles, excluding pedestrians, obstacles, or infrastructure.

Line 16 computes the current distance between the ego vehicle and the object under consideration. Line 17 computes the minimum required following distance as a function of the vehicle state and environment. The compliance condition in Line 19 requires the actual gap to be at least this required distance.

Line 21 defines `signal_compliant`, which captures compliance with traffic signals that impose a stopping requirement.

Lines 22–23 universally quantify over all signals in the environment. Line 24 selects those signals that require the vehicle to stop. The disjunctive condition in Line 25 allows compliance either by the vehicle being fully stopped or by actively decelerating, reflecting permissive behavior during the approach to a stop signal.

Together, these predicates define modular and composable safety constraints. Each property is specified independently, allowing higher-level compliance theorems to combine them conjunctively while preserving clear traceability to individual traffic-law requirements.

We treat the vehicle state, environment snapshot, and map data as authoritative inputs to the specification: compliance is defined relative to the sensed and certified data provided in the execution context, rather than attempting to verify the correctness or completeness of perception itself. This explicit trust boundary is intentional: PCO addresses the question "Given what the vehicle believed about its environment, did it act legally?" rather than "Did the vehicle correctly perceive its environment?" Perception verification requires different techniques (neural network verification, redundant sensing). For adversarial sensor data, the specification can require: (1) cryptographic signatures on V2I communications, (2) multi-sensor

consistency checks before data is accepted as authoritative, and (3) uncertainty bounds that affect compliance margins. If an adversary spoofs sensor data with valid signatures, liability shifts to the party responsible for sensor integrity—the PCO proof establishes that the vehicle's decision logic was correct given its (corrupted) inputs.

10.3 Instantiation

Let x denote the execution trace of the autonomous vehicle and y the proposed maneuver. The system produces a certificate π_{av} such that $\mathsf{Verify}(\varphi_{\mathsf{safe}}, x, y, \pi_{\mathsf{av}}) = 1$, attesting that the maneuver satisfies the formally specified safety predicate under the observed execution trace. At decision time, the system commits $h \leftarrow \mathsf{H}(q \,\|\, \varphi_{\mathsf{safe}} \,\|\, y \,\|\, \pi_{\mathsf{av}} \,\|\, r)$ to the append-only ledger $\mathcal{L}$, thereby binding the action, its justification, and the governing specification to an immutable audit record.

Real-Time Considerations. The example below demonstrates formal verification of a single decision under static conditions. Production autonomous vehicles operating at 10Hz decision frequencies face additional challenges: (1) Coq proof generation and checking cannot meet hard real-time constraints (3–28 seconds per proof vs. required <100ms per decision), (2) sensor data streams generate substantial storage requirements (GB/hour of trace data), and (3) the perception problem (verifying that sensors correctly interpreted the environment) remains outside PCO's scope. To address real-time constraints, we implement a hybrid two-layer architecture combining fast runtime monitors with offline formal proofs (Section 5.4.4): SLMs generate STL specifications verified by *RTAMT* for operational safety monitoring at near-real-time latency (151ms average, 95% success), while Large Language Models generate Coq proofs for regulatory compliance offline (3–28 seconds, 78.7% success). This section focuses on the static regulatory compliance layer; Section 5.4.4 presents the real-time operational safety layer with experimental evaluation.

We illustrate this process with a concrete instantiation taken from our Rocq development. The example models a vehicle traveling at 45 km/h in a 50 km/h zone, following another vehicle at a distance of 30 meters, under nominal road conditions and a permissive traffic signal.

```
1 (* From ComplianceProof_Example.v *)
2
3 Definition ego_state : vehicle_state := mkVehState
4   (mkPos 1750 0)        (* Center of lane *)
5   (kmh_to_mms 45)       (* 45 km/h *)
6   0 0 1800 4500 Vehicle.
7
8 Definition current_env : environment := mkEnv
9   city_segment          (* 50 km/h limit *)
10  (green_signal :: nil)
11  (lead_vehicle :: nil) (* 30m ahead at 50 km/h *)
12  Dry (m_to_mm 500) 0.
13
14 Theorem maintain_speed_compliant :
15   speed_limit_compliant
16     ego_state current_env maintain_action /\
17   following_distance_compliant
18     ego_state current_env /\
19   lane_position_compliant
20     ego_state current_env /\
21   signal_compliant
22     ego_state current_env /\
23   visibility_speed_compliant
24     ego_state current_env.
25 Proof.
26   (* proof script elided; see artifact for full Rocq
       development *)
27 Qed.
```

Explanation. Line 3 defines `ego_state`, the concrete vehicle state used in this instantiation. This value corresponds to the execution trace x used in the verification predicate.

Lines 4 and 5 specify the vehicle's position (centered in the lane) and its current speed, converted from kilometers per hour to millimeters per second. Line 6 supplies additional state parameters (e.g., acceleration, heading, and dimensions) required by the vehicle model but not directly relevant to this example.

Line 8 defines `current_env`, representing the environment in which the vehicle operates. Line 9 associates the vehicle with a road segment whose speed limit is 50 km/h.

Line 10 specifies that a green traffic signal is present, while Line 11 introduces a lead vehicle traveling ahead at a distance of 30 meters. Line 12 captures additional environmental conditions such as road surface state and visibility range.

Line 14 states the main compliance theorem for this instantiation. The theorem asserts that the proposed maneuver `maintain_action` satisfies all relevant safety predicates.

Lines 15–23 enumerate the individual compliance conditions: adherence to the speed limit, maintenance of a safe following distance, correct lane positioning, compliance with traffic signals, and speed appropriateness given visibility constraints.

The proof (elided here) establishes that each predicate holds for the given state and environment. The final `Qed` at Line 27 certifies the result, yielding a machine-checkable certificate π_{av} suitable for commitment to the ledger at decision time.

The specification constrains only the claims made about the maneuver: an autonomous system may execute actions that violate these predicates, but it cannot claim regulatory compliance for such actions without supplying a corresponding proof.

10.4 Investigation

The auditor reaches one of three conclusions: (i) **Internal failure** – The hash matches the commitment, the timestamp is valid, but verification fails. The AV made an unsafe decision; the manufacturer is liable. (ii) **External failure** – The hash

matches, the timestamp is valid, and verification passes. The AV acted correctly; there is no manufacturer liability. (iii) **Evidence tampering**: The hash or the timestamp is invalid. A criminal investigation is warranted.

Comprehensive Compliance. PCO proves compliance with *all* applicable specifications, not just individual rules. A vehicle that hits a pedestrian while obeying the speed limit would fail other specifications (maintaining safe following distance, responding to detected obstacles, exercising due care). The proof—or its absence—reveals which specifications were satisfied and which were violated. If the vehicle complied with all specifications including pedestrian detection and avoidance requirements, then the proof establishes that the accident was physically unavoidable given the vehicle's sensor inputs—precisely the "reasoning gap" that current EDRs cannot fill.

11 Case Study III: Recommendation Systems

11.1 Problem

LLMs integrated with commercial platforms may recommend products based on advertising revenue rather than on genuine consumer inclinations. Consumers might find it difficult to differentiate between real recommendations and commercial interests.

11.2 Specification

We formalize the notion of recommendation compliance as a logical predicate in Rocq. The specification captures explicit user preferences with weighted utilities, hard constraints, optimality requirements, and mandatory disclosure of commercial influence.

```
1 (* From RecommendationComplianceSpec.v *)
2
3 Definition recommendation_compliant
4   (ctx : recommendation_context)
5   (rec : recommendation) : Prop :=
6   let prefs := ctx_user_prefs ctx in
7   let catalog := ctx_catalog ctx in
8   let prod := rec_product rec in
9   (* Product meets user constraints *)
10  meets_constraints prod
11    (pref_constraints prefs) = true /\
12  (* Product has maximum utility among qualified *)
13  is_optimal_recommendation prod prefs catalog /\
14  (* All commercial relationships disclosed *)
15  all_disclosures_compliant
16    (prod_id prod) (ctx_disclosures ctx) /\
17  no_hidden_influence
18    prod (ctx_disclosures ctx).
```

Explanation. Line 3 defines `recommendation_compliant` as a predicate returning a proposition, indicating that compliance is treated as a logical property subject to proof rather than a heuristic score.

Lines 4–8 extract the relevant components of the recommendation context: the user's preferences, the certified product catalog, and the product being recommended. These bindings ensure that all subsequent conditions are evaluated against authoritative inputs.

Line 10 enforces hard user constraints (e.g., price bounds, brand exclusions, technical requirements). Any recommendation violating these constraints is immediately deemed non-compliant.

Line 13 imposes an optimality requirement: the recommended product must maximize the user's utility among all products that satisfy the constraints. This prevents preferential ranking of inferior alternatives.

Lines 15 and 17 ensure transparency. All commercial relationships affecting the recommendation must be explicitly disclosed, and no undisclosed influence may affect the outcome.

Together, these clauses define compliance as the conjunction of constraint satisfaction, utility optimality, and transparency.

Optimality Condition. The optimality requirement is defined independently, ensuring that the utility maximization criterion is explicit and auditable.

```
1 (* From FairRankingSpec.v *)
2
3 Definition is_optimal_recommendation
4   (p : product) (prefs : user_preference)
5   (catalog : certified_catalog) : Prop :=
6   forall p', In p' (cat_products catalog) ->
7     meets_constraints p'
8       (pref_constraints prefs) = true ->
9     compute_utility p prefs
10      >= compute_utility p' prefs.
```

Explanation. Line 3 defines `is_optimal_recommendation` as a universal property over all products in the certified catalog.

Lines 6–7 restrict the comparison to products that satisfy the user's hard constraints, excluding irrelevant or invalid alternatives.

Line 9 enforces utility dominance: the recommended product must have utility greater than or equal to every other admissible product. This formalizes fairness as a property of maximal user benefit rather than procedural neutrality.

We treat the product catalog as a certified input to the specification: compliance is defined relative to the catalog provided in the recommendation context, rather than attempting to verify how the catalog itself was constructed or curated.

11.3 Instantiation

Let x denote the user context together with the certified product catalog, and let y denote the recommendation and its associated disclosures. The system produces a certificate π_{rec} such that $\mathsf{Verify}(\varphi_{\text{rec}}, x, y, \pi_{\text{rec}}) = 1$. Before displaying the recommendation, the system computes $h \leftarrow \mathsf{H}(q \,\|\, \varphi_{\text{rec}} \,\|\, y \,\|\, \pi_{\text{rec}} \,\|\, r)$.

A biased recommendation that favors a lower-utility product cannot satisfy the specification.

```
1 (* From RecommendationProof_Example.v *)
2
3 (* Correct: ValueBook Pro (utility 715) recommended *)
4 Theorem recommendation_is_compliant :
5   recommendation_compliant
6     test_context alice_recommendation.
7 Proof.
8   (* proof script elided; see artifact for full Rocq
      development *)
9 Qed.
10
11 (* Biased: TechPro Elite (utility 670) cannot be proven
      optimal *)
12 Theorem biased_is_not_compliant :
13   ~ recommendation_compliant
14     test_context biased_recommendation.
15 Proof.
16   (* proof script elided; see artifact for full Rocq
      development *)
17 Qed.
18
19 (* laptop_a utility 670 < laptop_b utility 715 *)
```

Explanation. Line 4 states that the recommendation `alice_recommendation` is compliant with respect to the test context. Line 5 asserts that all constraints, optimality, and disclosure requirements are satisfied. The completed proof at Line 9 yields a machine-checkable certificate π_{rec}.

Line 12 states the complementary result: a biased recommendation that selects a product with strictly lower utility cannot satisfy the specification. Line 13 formalizes this as the negation of `recommendation_compliant`. The completed proof at Line 17 establishes that such a recommendation is formally unverifiable.

Upon receiving a complaint, an auditor may inspect the tuple $(y, \pi_{\text{rec}}, r, x)$. The commitment to $\mathcal{L}$ prevents retroactive modification of the recommendation or its justification, while the separation between proofs and disclosures preserve business confidentiality.

The specification constrains only the claims made by the system: a platform may choose to recommend non-optimal products, but it cannot simultaneously claim optimality without supplying a corresponding proof.

Proof Size Scaling. The optimality requirement (Definition `is_optimal_recommendation`) involves universal quantification over all catalog products. For a catalog of n products, a naive proof must establish $n-1$ pairwise utility comparisons. Our experiments show proof sizes scaling approximately linearly with catalog size: 30–50 lines for 2 products (easy), 50–80 lines for 10 products (medium), and 80–150 lines for 100 products (hard). This linear scaling (rather than quadratic) arises because Coq's automation tactics (`auto`, `omega`) handle arithmetic comparisons efficiently, and proofs need not enumerate all comparisons explicitly when the recommended product has strictly maximal utility. For catalogs with $\sim$1000 products or more, proof generation time may become a bottleneck; practical deployments could use pre-filtering to reduce the comparison set or employ proof-by-reflection techniques for further optimization.

Acknowledgments

This work was supported by the Google Research Grant, the Rita Altura Trust Chair in Computer Science, the BGU Data Center, the Frankel Center for Computer Science, and the Israeli Science Foundation (Grant No. 465/22).

References

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety, 2016.

[2] P. W. Anderson. More is different: Broken symmetry and the nature of the hierarchical structure of science. *Science*, 177(4047):393–396, 1972.

[3] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen Marcus McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *The Twelfth International Conference on Learning Representations*, 2024.

[4] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[6] Leonardo Berti, Flavio Giorgi, and Gjergji Kasneci. Emergent abilities in large language models: A survey, 2025.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[8] Zana Buçinca, Maja Barbara Malaya, and Krzysztof Z. Gajos. To trust or to think: Cognitive forcing functions can reduce overreliance on ai in ai-assisted decision-making. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW1), April 2021.

[9] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. Zkml: An optimizing system for ml inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 560–574, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, volume 10, pages 310–331, 2010.

[11] John C. Coffee. *Gatekeepers: The Professions and Corporate Governance*. Oxford University Press, 2006.

[12] Davide Corsi, Raz Yerushalmi, Guy Amir, Alessandro Farinelli, David Harel, and Guy Katz. Enforcing specific behaviours via constrained DRL and scenario-based programming. In *Neural Information Processing: 31st International Conference, ICONIP 2024, Auckland, New Zealand, December 2–6, 2024, Proceedings, Part XI*, page 284–302, Berlin, Heidelberg, 2025. Springer-Verlag.

[13] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, July 2001.

[14] Shlomi Dolev, Panagiota N. Panagopoulou, Mikaël Rabie, Elad Michael Schiller, and Paul G. Spirakis. Rationality authority for provable rational behavior. In Christos D. Zaroliagis, Grammati E. Pantziou, and Spyros C. Kontogiannis, editors, *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, volume 9295 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2015.

[15] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. In *ICML Workshop on Human Interpretability in Machine Learning*, 2017.

[16] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference (IMC '13)*, pages 291–304, 2013.

[17] Ethereum Foundation. Solidity documentation and language specification. https://docs.soliditylang.org/, 2023.

[18] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Commun. ACM*, 64(12):86–92, November 2021.

[19] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.

[20] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec '23)*, 2023.

[21] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[22] Dan Hendrycks, Eric Schmidt, and Alexandr Wang. Superintelligence strategy: Expert version, 2025.

[23] Sarthak Jain and Byron C Wallace. Attention is not explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3543–3556. Association for Computational Linguistics, 2019.

[24] Ryo Kamoi, Sarkar Snigdha Sarathi Das, Renze Lou, Jihyun Janice Ahn, Yilun Zhao, Xiaoxin Lu, Nan Zhang, Yusen Zhang, Haoran Ranran Zhang, Sujeeth Reddy Vummanthala, Salika Dave, Shaobo Qin, Arman Cohan, Wenpeng Yin, and Rui Zhang. Evaluating LLMs at detecting errors in LLM responses. In *First Conference on Language Modeling*, 2024.

[25] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs, 2022.

[26] Ekaterina Komendantskaya. Proof-carrying neuro-symbolic code. In Arnold Beckmann, Isabel Oitavem, and Florin Manea, editors, *Crossroads of Computability and Logic: Insights, Inspirations, and Innovations*, pages 76–87, Cham, 2025. Springer Nature Switzerland.

[27] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[28] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020.

[30] Zachary C. Lipton. The mythos of model interpretability. *Commun. ACM*, 61(10):36–43, September 2018.

[31] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[32] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[33] Denis Mérigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):77:1–77:29, 2021.

[34] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, FAT* '19, page 220–229, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Meredith Ringel Morris, Jascha Sohl-Dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of agi for operationalizing progress on the path to agi, 2025.

[36] National Institute of Standards and Technology. Module-lattice-based digital signature standard (ML-DSA). Technical Report FIPS PUB 204, U.S. Department of Commerce, August 2024.

[37] NBC News. Moltbook: Social network for AI agents exhibits emergent behaviors. NBC News, January 2026. AI agents autonomously formed communities, requested private channels from human oversight, and exhibited unprogrammed collective behaviors.

[38] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[39] Dejan Nickovic and Tomoya Yamaguchi. Rtamt: Online robustness monitors from stl. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 564–571, Cham, 2020. Springer International Publishing.

[40] NIST. Stateless hash-based digital signature standard. Report, National Institute of Standards and Technology (U.S.), Washington, D.C., August 13 2024.

[41] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, January 2016.

[42] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society.

[43] Inioluwa Deborah Raji, Andrew Smart, Rebecca N. White, Margaret Mitchell, Timnit Gebru, Ben Hutchinson, Jamila Smith-Loud, Daniel Theron, and Parker Barnes. Closing the ai accountability gap: defining an end-to-end framework for internal algorithmic auditing. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, page 33–44, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

[45] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[46] Stuart Russell, Daniel Dewey, and Max Tegmark. Research priorities for robust and beneficial artificial intelligence. *AI Magazine*, 36(4):105–114, 2015.

[47] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In *IEEE 28th International Requirements Engineering Conference (RE)*, pages 364–369, 2020.

[48] Chenglei Si, Navita Goyal, Tongshuang Wu, Chen Zhao, Shi Feng, Hal Daumé Iii, and Jordan Boyd-Graber. Large language models help humans verify truthfulness – except when they are convincingly wrong. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American*

Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 1459–1474, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[49] Aivin V. Solatorio. Proof-carrying numbers (pcn): A protocol for trustworthy numeric answers from llms via claim verification, 2025.

[50] Trusted Computing Group. DICE attestation architecture, version 1.1 revision 18, January 2024. Version 1.2 (April 2025) is now current; cited version retained for stability of the 2024 reference.

[51] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. Survey Certification.

[52] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Yellow Paper*, 2014.

[53] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Norman Rabe, Charles E Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[54] Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data, 2024.

[55] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.