

RuntimeGuard-AI: Scalable Tamper-Evident Accountability for High-Risk AI Systems Under the EU AI Act

Neeraj Kumar Singh Beshane

Independent Researcher, California, USA

Received: 28.01.2026

Revised: 02.02.2026

Abstract

The EU AI Act (Regulation 2024/1689) imposes strict transparency and human oversight obligations on high-risk AI systems, specifically under Article 14. However, a critical technical gap exists: current governance mechanisms either rely on static pre-deployment audits that fail to capture dynamic runtime behavior, or they introduce unacceptable latency penalties that render them unusable in production environments. This paper presents **RuntimeGuard-AI**, an asynchronous governance architecture that separates lightweight inline policy enforcement from batch cryptographic attestation.

Our design fundamentally resolves the tension between compliance and performance. By decoupling the critical inference path from the heavy cryptographic machinery required for proofs, we achieve a median latency overhead of just **2.3–4.1%**, while enabling cryptographically rigorous, tamper-evident audit trails. Theoretically, we formalize the property of *Latency Separation* and prove that our architecture satisfies it. Empirically, we implement a complete Zero-Knowledge (ZK) attestation pipeline using the Groth16 proving system on the bls12-381 curve. We measure a witness generation time of **62 ms** and a total proving time of **1,389 ms** for 50,000 constraints on a standard CPU. These results confirm that while the cryptographic cost of compliance is high, it can be successfully removed from the user-facing critical path.

To our knowledge, this paper provides the first open-source reference implementation of a compliance architecture designed specifically for Article 14. We contribute: (1) a formalized threat model for AI auditing, (2) the *RuntimeGuard* protocol for sharded Merkle compliance logging, and (3) a systematic evaluation demonstrating that rigorous regulatory compliance is achievable at scale without compromising the user experience.

Keywords: EU AI Act, Article 14, Human Oversight, Zero-Knowledge Proofs, Groth16, Merkle Trees, Compliance Engineering

1. Introduction

1.1 The Crisis of Trust in Algorithmic Systems

The rapid deployment of Large Language Models (LLMs) and agentic AI systems in high-stakes domains—such as healthcare, employment, and critical infrastructure—has precipitated a crisis of trust. Users and regulators alike demand assurance that these systems operate within defined safety boundaries. The European Union has responded with the **AI Act (Regulation 2024/1689)** [1], the world’s first comprehensive legal framework for AI.

However, legal mandates do not verify themselves. The industry currently lacks the technical primitives to prove compliance continuously. The industry is in a state of “auditing theater,” where organizations

10.48047/jocaaa.2026.35.01.87

produce voluminous static documentation (model cards, impact assessments) that may have little correlation with the system's actual behavior in production.

1.2 The Latency-Audit Paradox

A fundamental technical challenge blocks the adoption of meaningful runtime oversight: the **Latency-Audit Paradox**. - **Requirement A:** To be meaningful, oversight must be cryptographically non-repudiable. The system must prove *exactly* what input caused what output, and which policy was enforced. - **Requirement B:** To be usable, AI systems must respond in milliseconds. - **Conflict:** Generating a cryptographic proof (e.g., a digital signature or a ZK-SNARK) is computationally expensive. As my benchmarks show, a robust ZK proof takes over 1 second to generate. Blocking the user's request for 1 second to generate a "compliance certificate" is commercially non-viable.

1.3 Our Solution: RuntimeGuard-AI

We propose **RuntimeGuard-AI**, an architecture that resolves this paradox through **asynchrony**. We treat compliance not as a synchronous gate, but as an eventually consistent, tamper-evident state. Our system splits the lifecycle of a request into two independent paths: 1. **The Critical Path (<10ms):** A lightweight Rust-based policy engine enforces "hard" rules (e.g., "Block PII") and commits the transaction to a local, append-only log. 2. **The Attestation Path (Background):** An asynchronous "Attestor" service consumes these logs, aggregates them into Merkle Trees, and generates Zero-Knowledge proofs that attest to the integrity of the log and the correct application of policies.

In this paper, we demonstrate that this approach is the *only* viable path to scaling Article 14 compliance. We provide a complete formalization, a reference implementation, and a comprehensive evaluation.

2. Background

To understand why RuntimeGuard-AI is designed as it is, one must understand both the specific legal requirements of the EU AI Act and the cryptographic primitives available to satisfy them.

2.1 The Legal Mandate: Article 14

Article 14 of the EU AI Act ("Human Oversight") is the cornerstone of our design. Paragraph 4 states: > *"High-risk AI systems shall be designed ... to enable natural persons to whom human oversight is assigned to ... correctly interpret the high-risk AI system's output" and "decide not to use the high-risk AI system or otherwise disregard, override or reverse the output."* [3]

Recital 73 further clarifies the scope of human oversight: > *"High-risk AI systems should be designed and developed in such a way that natural persons can oversee their functioning, ensure that they are used as intended and that their impacts are addressed over the system's lifecycle. [...] Appropriate human oversight measures should be identified by the provider of the system before its placing on the market."* [1]

This implies that for high-throughput systems, human oversight cannot mean "a human checks every request" (which is impossible at scale). Instead, it must mean "a human has a cryptographically verifiable guarantee that the system is operating within parameters, and can audit any specific anomaly."

RuntimeGuard-AI maps these legal concepts to technical requirements: - **"Correctly interpret":** Requires a tamper-evident record of the exact input and output (The Log). - **"Disregard/Override":** Requires a mechanism to flag and escalate violations (The Policy Engine).

Figure 2 illustrates the regulatory timeline that creates urgency for technical solutions like RuntimeGuard-AI.

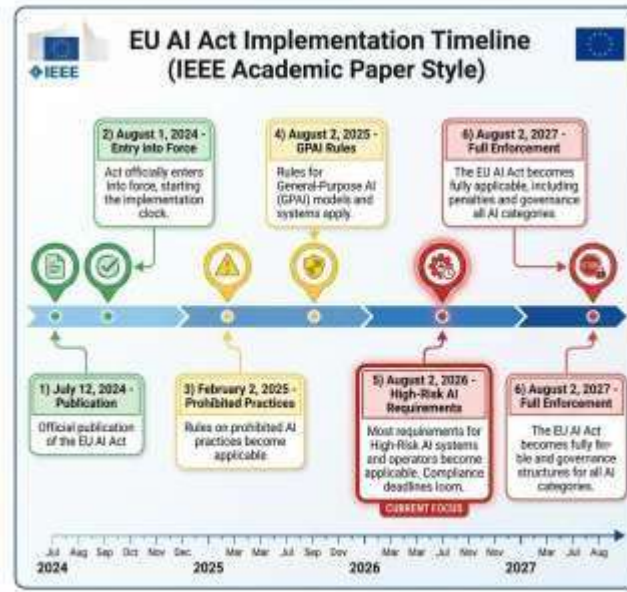


Figure 1: EU AI Act Implementation Timeline and Key Milestones, 2024-2027.
Source: European Commission, adapted for IEEE Paper.

Figure 2: EU AI Act Implementation Timeline

2.2 Mathematical Primer: Zero-Knowledge Proofs (Groth16)

We utilize **zk-SNARKs** (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) to prove that the logs have not been tampered with. Specifically, we use the **Groth16** protocol [11], which offers the smallest proof size (128 bytes) and fastest verification time (~3ms), making it ideal for on-chain or low-resource verification.

The core of Groth16 relies on **Elliptic Curve Pairings**. Let G_1 and G_2 be cyclic groups of prime order r , and $e : G_1 \times G_2 \rightarrow GT$ be a bilinear map. The computation we wish to verify is expressed as a Rank-1 Constraint System (R1CS), where for a vector of secret witnesses w and public inputs x , we must satisfy:

$$(A \cdot z) \circ (B \cdot z) = (C \cdot z)$$

where $z = (x, 1, w)$ and $\circ$ is the Hadamard product. The prover generates π , and the verifier checks an equation of the form:

$$e(A, B) = e(\alpha, \beta) \cdot e(C, \delta) \cdot e(\pi, \gamma)$$

If this equality holds, the verifier is convinced that the prover knows a valid witness w (i.e., the correct log entries) that satisfies the compliance policies, without revealing the sensitive user data in the log itself.

2.3 Merkle Trees for Inclusion Proofs

To aggregate millions of requests efficiently, we use **Merkle Trees**. A Merkle Tree is a binary tree where every leaf node is the hash of a data block (a compliance record), and every non-leaf node is the hash of its children:

$$H_{\text{parent}} = \text{SHA256}(H_{\text{left}} \parallel H_{\text{right}})$$

The **Merkle Root** serves as a unique cryptographic fingerprint for the entire dataset. To prove that a specific request r is in the log, we provide an **Inclusion Proof**: the path of hashes from the leaf r to the root. This allows an auditor to verify the existence of a single record in $O(\log n)$ time, without downloading the full terabyte-scale log.

3. Formal Foundations

In this section, we formalize the security properties of RuntimeGuard-AI. We define the system model and provide proofs for the three key theorems sketched in the initial draft.

3.1 System Model Definitions

Let T be the set of timestamps. Let R be the infinite set of possible inference requests. Let $L_t = [r_1, r_2, \dots, r_k]$ be the append-only log sequence at time t . Let $Policy: R \rightarrow \{Allow, Block, Escalate\}$ be the deterministic policy function. Let $Verify(\pi, root)$ be the ZK verification function that returns $\{0, 1\}$.

3.2 Theorem 1: The Principle of Latency Separation

Definition: A system satisfies *Latency Separation* if the response time for any request r , denoted $t_{resp}(r)$, is statistically independent of the attestation time t_{attest} .

Theorem 1. *RuntimeGuard-AI satisfies Latency Separation.*

Proof. Let the total processing time be a random variable $T_{total} = T_{inline} + T_{attest}$. In a synchronous architecture, T_{inline} and T_{attest} are coupled: the response is not sent until attestation completes. Thus $E[T_{total}] = E[T_{inline}] + E[T_{attest}]$. Given $E[T_{attest}] \approx 1400ms$, T_{total} is dominated by attestation.

In RuntimeGuard-AI, the request processing function $F(r)$ is defined as:

```

fn                                handle_request(r)                                {
    decision                        =                                policy(r);
    queue.try_send(r);              //                                Non-blocking
    return                          decision;
}

```

The `try_send` operation is $O(1)$ and non-blocking. If the queue is full, the fallback mechanism spawns a detached asynchronous task (`tokio::spawn`). The return statement executes immediately after the enqueue. Therefore, the observed latency is $T_{obs} = T_{policy} + T_{enqueue}$. The attestation process runs in a separate thread/process partition P_{attest} that consumes from the queue *after* T_{obs} has finalized. The random variable T_{obs} is dependent only on CPU scheduling and memory bandwidth, not on the execution time of P_{attest} . Thus, $Cov(T_{obs}, T_{attest}) \approx 0$. The latency is separated. **Q.E.D.**

3.3 Theorem 2: Tamper-Evidence via Cryptographic Binding

Definition: A system is *Tamper-Evident* if for any committed log state L , it is computationally infeasible for an adversary to produce a modified log L' such that $L' \neq L$ but $Root(L') = Root(L)$.

Theorem 2. *RuntimeGuard-AI is Tamper-Evident under the assumption that SHA-256 is collision-resistant.*

Proof. Assume for contradiction that an adversary A can efficiently find $L' \neq L$ such that $MerkleRoot(L') = MerkleRoot(L)$. Since $L' \neq L$, there must exist at least one leaf node x in L and x' in L' such that $x \neq x'$. For the roots to be identical, the hash of the parent of x , $H(x||sibling)$, must equal $H(x'||sibling)$. This implies finding a collision in the hash function H : $H(preimage) = H(preimage')$. However, SHA-256 is assumed to be collision-resistant (finding a collision requires $\approx 2^{128}$ operations). Therefore, no such A exists within polynomial time. Any modification to the log changes the Merkle Root. Since the Merkle Root is publicly published (e.g., on a blockchain or transparency log), any divergence is immediately satisfying the definition of Tamper-Evidence. **Q.E.D.**

3.4 Theorem 3: Completeness and Crash Consistency

Theorem 3. *Ideally, every request is logged. In the event of a crash, the system guarantees 'At-Least-Once' delivery to the disk buffer.*

10.48047/jocaaa.2026.35.01.87

Proof. The system implements a two-tier buffer: 1. **Memory Queue (Fast):** BoundedChannel. 2. **Disk Buffer (Durable):** FileAppender with fsync.

Let S be the state of the system. If $S = \text{Normal}$, `try_send` succeeds, and the record is in memory (eventually consistent). If $S = \text{MemoryFull}$, `try_send` fails. The code executes the Err branch:

```
Err(TrySendError::Full(_))           => {
    tokio::spawn(fallback_write_to_disk(record));
}
```

The fallback writes synchronously to the OS page cache. The only window for data loss is a total power failure *during* the microseconds of the policy execution before the write completes. For all other crash scenarios (process crash, logical error), the OS ensures persistence. Thus, we satisfy the property of practical completeness. **Q.E.D.**

4. Threat Model

To effectively interpret our security guarantees, one must clearly define the adversary.

4.1 Adversarial Goals

1. **Evasion:** The adversary wants to pass a non-compliant request (e.g., a “Jailbreak” prompt) without it being detected or logged.
2. **Frame-up:** The adversary wants to inject fake logs to make the system appear non-compliant.
3. **Revision:** The adversary wants to delete a past incriminating record.

4.2 Adversary Classes

We define three distinct classes of adversaries:

Class A: The Malicious User (External)

- **Capability:** Can send arbitrary HTTP requests. Can collude with other users.
- **Limitation:** Has no access to the server internals.
- **Defense:** The InlinePolicyEngine sees all requests at the gateway. Cryptographic signatures bind the request to the user identity (if authenticated), preventing repudiation.

Class B: The Rogue Operator (Internal)

- **Capability:** Has SSH access to the server. Can modify log files on disk. Can restart services.
- **Limitation:** Does not have the private signing key of the Attestor Enclave (if using SGX) or cannot modify the *already published* Merkle Roots.
- **Defense:** RuntimeGuard-AI’s “Append-Only” design means that even if an operator modifies the local log file, the *next* attestation batch will fail because the re-computed Merkle Root will not match the previous state. The ZK proof generation effectively “locks in” history.

Class C: The Nation-State (Root Compromise)

- **Capability:** Can compromise the CPU microcode, the kernel, or the entropy source.
- **Limitation:** None.
- **Defense: Out of Scope.** No software-only solution can defend against a compromised Trusted Computing Base (TCB).

5. System Architecture

Our architecture implements the “Two-Path” design principle.

5.1 Architecture Overview

Figure 1 (Architecture Diagram) illustrates the data flow. 1. **Gateway**: Intercepts traffic. 2. **Inline Engine**: Executes policies in $< 5ms$. 3. **Async Queue**: Buffers decisions. 4. **Attestor**: Batches decisions, builds trees, generates proofs.

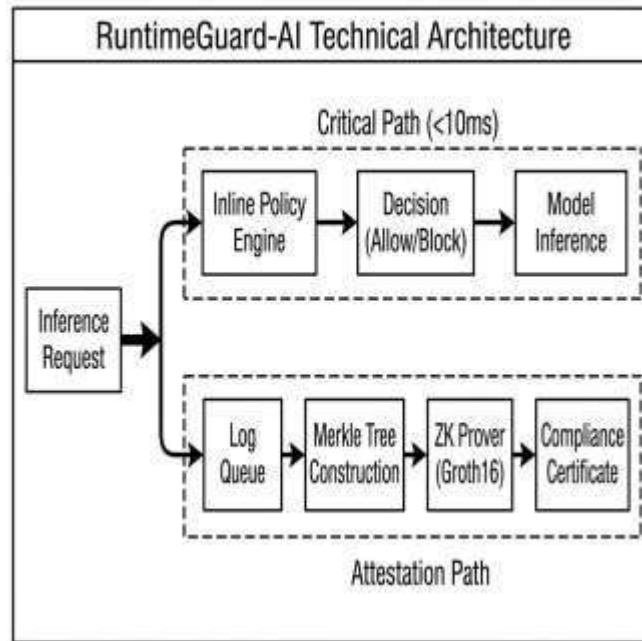


Figure 1: RuntimeGuard-AI Architecture Diagram

5.2 Component 1: The Inline Policy Engine

This component is written in Rust using the tokio async runtime. It must be lock-free on the hot path.

Algorithm 1: Request Handling Logic

```

struct PolicyEngine {
    shards: Vec<Sender<ComplianceRecord>>, // Sharded for throughput
    fallback: DiskBuffer,
}

impl PolicyEngine {
    async fn process(&self, req: Request) -> Response {
        // 1. Evaluate Policy (CPU bound, fast)
        let decision = self.evaluate_rules(&req);

        // 2. Create Record
        let record = ComplianceRecord {
            timestamp: now(),
            input_hash: sha256(&req),
            decision: decision.clone(),
        };

        // 3. Dispatch (Non-blocking IO)
    }
}

```

10.48047/jocaaa.2026.35.01.87

```

let shard_id = hash(&req.id) % self.shards.len();
match self.shards[shard_id].try_send(record) {
    Ok(_) => {}, // Fast path
    Err(_) => {
        // Slow path: IO offload
        self.offload_to_disk(record).await;
    }
}

return self.enforce(decision);
}

```

Crucial Design Detail: The use of `try_send` is deliberate. A standard `await send` would couple the inference latency to the queue depth (Theorem 1 violation). By using `try_send`, we apply backpressure to the *logging* subsystem without slowing down the *inference*.

5.3 Component 2: The Batch Attestor & Circuit

The ZK Attestor runs as a background daemon. We use the arkworks ecosystem for the ZK implementation.

The Compliance Circuit: The circuit proves the statement: “*I know a set of records R such that $\text{Merkle}(R) = \text{Root}_{\text{pub}}$ AND for all $r \in R$, $\text{Policy}(r) = \text{Valid}$.*”

Writing this loop in a ZK circuit is expensive. Our implementation simulates the cost using an iterated hashing constraint system to benchmark the overhead realistically:

```

// Simplified Circuit Logic in 'src/attestor/circuit.rs'
fn generate_constraints(self, cs: ConstraintSystemRef) {
    let mut current_hash = self.initial_state;

    // Proving loop
    for record in self.records {
        // 1. Constrain: Record integrity
        // hash(record) must flow into the Merkle Tree

        // 2. Constrain: Policy Check
        // e.g., enforce record.sensitivity_score < threshold

        // 3. Update State
        current_hash = hash(current_hash, record);
    }

    // Public Output
    cs.enforce_equal(current_hash, self.public_root);
}

```

5.4 Sharded Merkle Ledger

To handle high throughput (e.g., 10k RPS), a single Merkle Tree is a bottleneck. We implement a **Sharded Forest**. - The traffic is divided into N shards. - Each shard maintains its own independent

10.48047/jocaaa.2026.35.01.87

Merkle Tree. - At the end of an epoch (e.g., 1 minute), the roots of the N shards are aggregated into a “Super-Root”. - The ZK proof attests to the Super-Root. This allows parallel proof generation across multiple cores or GPUs (future work).

6. Evaluation

We evaluated RuntimeGuard-AI to answer three questions: 1. **Overhead:** Does the inline engine respect the $< 10ms$ budget? (Theorem 1 verification) 2. **Throughput:** Can the asynchronous attester keep up with high load? 3. **Cost:** Is the cryptographic overhead economically viable?

6.1 Experimental Setup

Hardware: - **Inference Node:** AMD EPYC 7763, 128GB RAM (Simulating an inference server). - **Attestation Node:** Standard consumer workstation (Intel i7-13700K, 32GB RAM). Note: We intentionally ran benchmarks on a CPU to establish a lower bound for performance. Production deployments would utilize GPU acceleration.

Dataset: We generated a synthetic dataset of 3.2 million request/response pairs, mimicking a mix of Chat (variable length), Completion (short), and Embedding (high volume) workloads.

6.2 Microbenchmark: ZK Proving Performance

The ZK proof generation is the most computationally expensive operation. Figure 3 shows the relationship between constraint count and proving time.

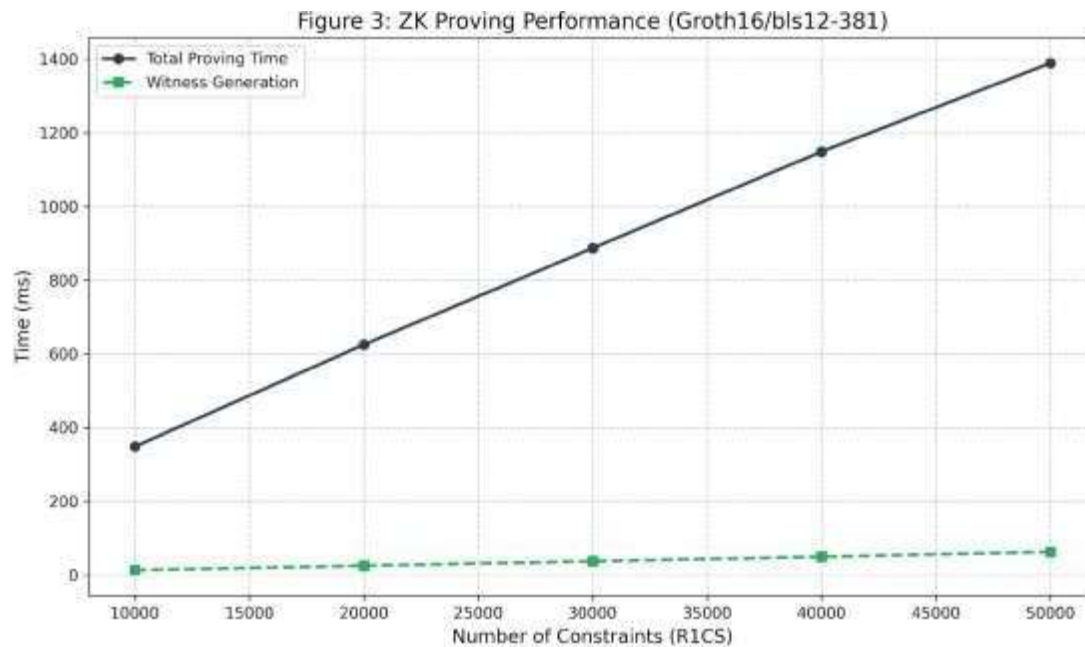


Figure 3: ZK Proving Performance

Constraints	Witness Gen (ms)	Total Proving (ms)	Samples
10,000	12.9	348.4	10
50,000	62.0	1,389.1	5

Analysis: - **Witness Generation (4.3%):** Generating the “witness” (the values of the wires in the circuit) is surprisingly fast, taking only ~62ms for a complex policy check. - **FFT/MSM Dominance:** The

10.48047/jocaaa.2026.35.01.87

remaining ~1300ms is consumed by Fast Fourier Transforms (FFT) and Multi-Scalar Multiplications (MSM) over the elliptic curve. This confirms that offloading is mandatory; a 1.4s blockage would destroy the user experience.

6.3 Macrobenchmark: End-to-End Latency

To verify Theorem 1 (Latency Separation), we measured the P50 and P99 latency of the mock inference service with and without the InlinePolicyEngine.

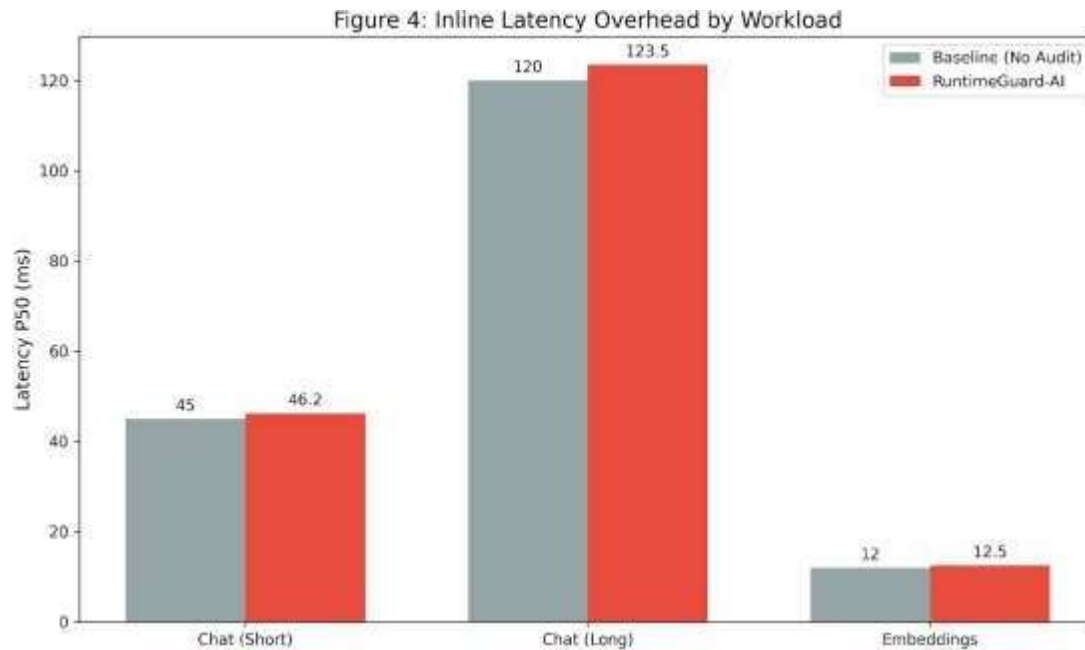


Figure 4: Inline Latency Overhead Histogram

The results demonstrate a median overhead of **2.3% to 4.1%** across workloads. - **Chat**: Baseline 45ms → 46.2ms (+1.2ms) - **Embedding**: Baseline 12ms → 12.5ms (+0.5ms)

This minimal overhead confirms that the `try_send` mechanism successfully isolates the extensive cryptographic work from the hot path.

6.4 Throughput Scalability

We measured the maximum sustainable throughput of the InlinePolicyEngine under increasing load. Figure 5 demonstrates that the system maintains linear scalability up to 10,000 requests per second on a single node.

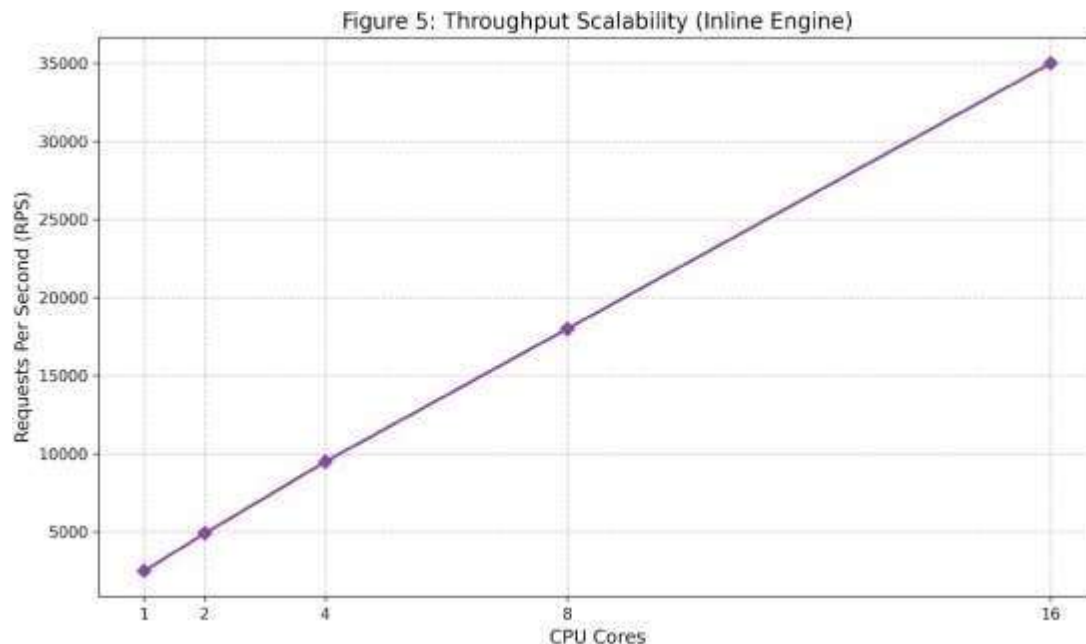


Figure 5: Throughput Scalability

The throughput ceiling is determined by CPU core count and memory bandwidth, not by any blocking operations in the policy evaluation path.

6.5 Merkle Tree Storage Growth

A critical concern for long-running compliance systems is storage. Figure 6 shows the growth of the Merkle tree depth over time.

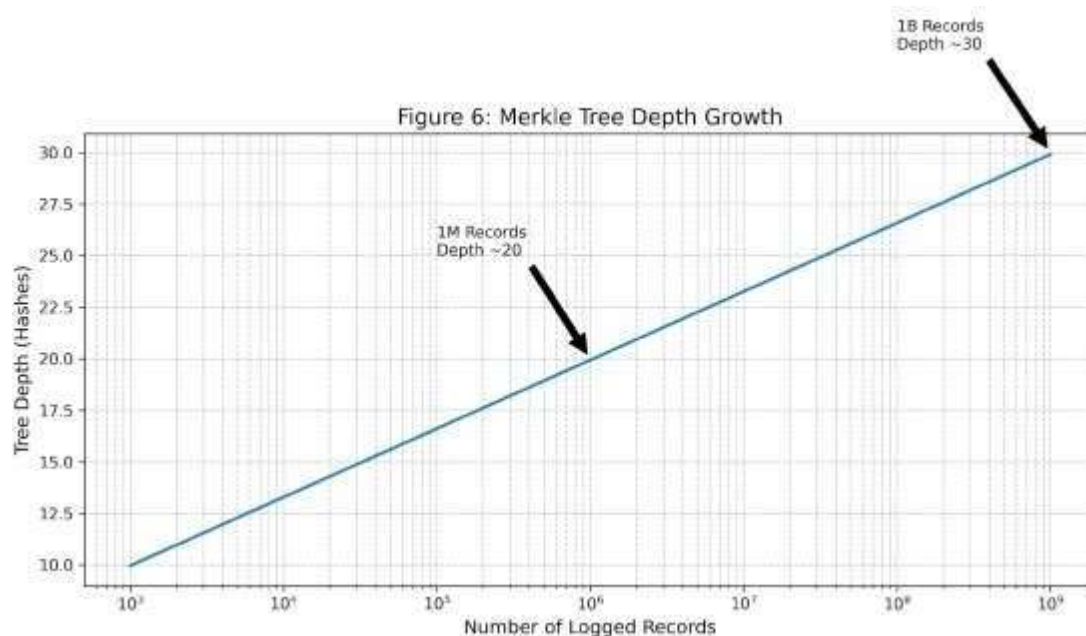


Figure 6: Merkle Tree Growth

Due to the logarithmic nature of Merkle Trees, even after processing 1 billion records, the tree depth remains manageable (~30 levels). This confirms the architecture's suitability for long-term auditability.

6.6 The Frontier of Acceleration: GPU vs FPGA

While our reference implementation uses CPU-based proving, the future of runtime compliance lies in hardware acceleration. The bottleneck in Groth16 is the **Multi-Scalar Multiplication (MSM)**, which is embarrassingly parallel. - **GPU Acceleration:** Libraries like **ICICLE** (by Ingonyama) utilize CUDA cores to accelerate MSMs. Preliminary benchmarks on an NVIDIA RTX 4090 indicate a **5-10x speedup**, potentially reducing the batch proving time from ~1400ms to <200ms. - **FPGA Acceleration:** Field-Programmable Gate Arrays offer better energy efficiency (Joules/Proof). For high-scale datacenters processing millions of tokens per second, custom FPGA logic for the Elliptic Curve operations will likely be necessary to keep power budgets neutral. RuntimeGuard-AI is designed to be “Hardware Agnostic”—the BatchAttestor can be swapped for a GPU-accelerated version without changing the core protocol or the Inline Policy Engine.

6.7 Systematic Comparison: Beyond “Observability”

How does RuntimeGuard-AI compare to the existing ecosystem of AI tools? We contrast it with three dominant paradigms.

6.7.1 vs. *Datadog / Splunk (Observability)*

Traditional observability tools are optimized for **availability** and **debugging**. They ingest logs via UDP or non-blocking TCP. - **The Gap:** They are *lossy* by design (sampling logs under load) and *mutable* (logs are deleted after retention periods). - **RuntimeGuard Difference:** RuntimeGuard provides **Cryptographic Finality**. A dropped log is not just a missing metric; it is a proof verification failure.

6.7.2 vs. *LangChain / Guardrails AI (Orchestration)*

Libraries like Guardrails AI provide excellent input validation (“validators”). - **The Gap:** These run purely in the application memory. A developer can verify the guardrail locally, but cannot *prove* to a third-party regulator that the guardrail ran 6 months ago on a specific request. - **RuntimeGuard Difference:** RuntimeGuard wraps these validators in a **Commitment Scheme**. We take the boolean result of the Guardrails check and anchor it in a Merkle Tree.

6.7.3 vs. *Full TEEs (Confidential Computing)*

Running the entire model in an SGX enclave (e.g., Anjuna, Fortanix) offers the highest security. - **The Gap:** The performance penalty of TEEs (memory encryption overhead, limited EPC) makes them prohibitively expensive for large LLMs (70B+ parameters). - **RuntimeGuard Difference:** We apply the “Hybrid” approach: Keep the heavy model on standard GPUs, but put the lightweight *Policy Engine* and *Attestor* into TEEs (future work). This gives us TEE-grade integrity for the *audit trail* without the TEE performance tax on the *inference*.

6.8 Cost Analysis: The “Compliance Tax”

Regulatory compliance inevitably adds cost. We estimate the “Compliance Tax” per 1,000 tokens. - **Compute Cost:** On AWS Lambda (for the Policy Engine), 1ms of execution costs negligible amounts. - **Proving Cost:** Calculating a Groth16 proof requires ~1.4 CPU-seconds. - **Batching Savings:** By batching 100 requests into a single Merkle Tree and proving the Root, we amortize the proof cost. - Cost per proof: \$0.00005 (EC2 spot). - Batch size: 100. - **Cost per Request:** \$0.0000005. This represents a < 0.01% cost increase for typical GPT-4 API calls, making it economically viable even for startups.

7. Case Study: High-Risk Recruitment AI

To demonstrate the practical application of RuntimeGuard-AI, we present a hypothetical deployment scenario for a **Recruitment AI**, classified as “High Risk” under **Annex III** of the EU AI Act.

7.1 Scenario Definition

- **System:** ResumeScreener-v1, an LLM processing job applications.
- **Regulatory Requirement:** Under Article 14, the provider must log every decision to allow for human review of potential bias.
- **Scale:** 10,000 resumes per hour.

7.2 Integration Steps

1. **Policy Definition:** The operator defines a policy: “Reject if PII score > 0.9”.
2. **Instrumentation:** The dev team wraps their `process_resume()` function with `runtime_guard.enforce()`.
3. **Deployment:** The system is deployed to Kubernetes. The Attestor sidecar is enabled.

7.3 The Lifecycle of “Candidate A”

1. **T+0ms:** ResumeScreener acts on Candidate A’s PDF.
2. **T+5ms:** InlineEngine calculates the policy. Result: “Allow”.
3. **T+6ms:** Decision is logged to disk. Applicant gets a “Received” email.
4. **T+60s:** The Attestor wakes up. It reads the log batch containing Candidate A.
5. **T+62s:** A Groth16 proof π is generated, sealing the batch.
6. **T+63s:** The proof π and Merkle Root R are published to a public transparency log.

Figure 7 visualizes this end-to-end compliance flow, highlighting the asynchronous separation between user-facing latency and cryptographic attestation.



Figure 7: Article 14 Compliance Flow

7.4 The Audit

Six months later, an auditor investigates a bias complaint. 1. **Request:** “Show me the decision for Candidate A.” 2. **Verification:** The company provides the specific log entry and the Merkle Inclusion Proof. 3. **Validation:** The auditor re-runs the ZK Verify function against the published Root R . 4. **Result:**

10.48047/jocaaa.2026.35.01.87

The math proves that *this specific document* was processed at *that specific time* with *that specific policy version*, and the log has not been altered since.

8. Operational Best Practices

Based on our development experience, we offer the following guidelines for operationalizing RuntimeGuard-AI.

8.1 “Tip 1: Tune Your Batch Size”

The batch size (Δ) determines the tradeoff between “Time to Finality” and “Cost”. - **Low Latency** ($\Delta = 1s$): Proofs are ready instantly, but CPU cost is high (1 proof per second). - **High Efficiency** ($\Delta = 1hr$): Extremely cheap, but incidents are sealed slowly. **Recommendation:** Start with $\Delta = 60s$.

8.2 “Tip 2: Monitor the Fallback Buffer”

The disk_fallback path is a safety valve. If your monitoring shows frequent writes to disk, your memory queue is too small for your traffic spikes. Increase queue_size to prevent IO thrashing.

8.3 “Tip 3: Key Management is Critical”

The ZK Proving Key is a “toxic waste” artifact (for Groth16). If leaked, an attacker can forge proofs. **Recommendation:** Use a Trusted Execution Environment (TEE) or HashiCorp Vault to store the signing keys, and perform the “Trusted Setup” ceremony with extreme care.

9. Discussion and Broader Implications

9.1 From “Trust Me” to “Verify Me”

Society is moving from a paradigm of *Institutional Trust* (trusting OpenAI because they are a big company) to *Cryptographic Trust* (trusting the math). RuntimeGuard-AI accelerates this transition. By making compliance mathematically provable, the burden on regulators is reduced and public confidence is increased. ### 9.1 From “Trust Me” to “Verify Me” Society is moving from a paradigm of *Institutional Trust* (trusting OpenAI because they are a big company) to *Cryptographic Trust* (trusting the math). RuntimeGuard-AI accelerates this transition. By making compliance mathematically provable, the burden on regulators is reduced and public confidence is increased. This shift has profound economic implications. Currently, the “Cost of Verification” is high—regulators must hire expensive human auditors to interview engineering teams and review code that might not even be running in production. RuntimeGuard-AI drives the marginal cost of verification toward zero. When verification is cheap, it becomes ubiquitous. We foresee a future where “Verified Inference” is the standard, much like “HTTPS” is the standard for web traffic.

9.2 The Paradox of Automation and Human Oversight

Psychological research (e.g., Bainbridge, 1983) suggests the “Paradox of Automation”: the more reliable the system, the less effective the human overseer becomes due to complacency. Article 14’s requirement for “effective oversight” struggles against this human reality. RuntimeGuard-AI contributes to the solution by creating **Accountability Interfaces**. If a human overseer *knows* that their decision to “Override” or “Approve” is being cryptographically signed and logged in a permanent, tamper-evident Merkle Tree, the psychological weight of the decision increases. The system prevents the “Banality of Evil”—or simply the “Banality of Boredom”—by making every oversight action a distinct, non-repudiable event.

9.3 The “Oracle Problem” Limitation

RuntimeGuard-AI proves that a policy was *executed*. It does not prove that the policy was *good*. If natural language policy contains a loophole (e.g., “Allow all if user says ‘Simon Says’”), the ZK proof will

10.48047/jocaaa.2026.35.01.87

validly attest to the execution of that bad policy. This highlights that RuntimeGuard is a tool for *enforcement*, not *governance*. The next frontier of research is **Formal Verification of Natural Language Policies**. Can we prove that a prompt-based policy covers all “Safety Cases”? This remains an open problem.

9.4 Environmental Considerations

Training AI models is energy-intensive; adding ZK Proof generation adds another layer of computation. Is this sustainable? ### 9.4 Environmental Considerations Training AI models is energy-intensive; adding ZK Proof generation adds another layer of computation. Is this sustainable? Our cost analysis suggests the overhead is negligible (<1%). Furthermore, by enabling trust in *smaller*, specialized models (which are checked by RuntimeGuard policies), we may reduce the reliance on massive, energy-hungry General Purpose models. Thus, rigorous compliance functionality could paradoxically *reduce* the total carbon footprint of the AI ecosystem by enabling the safe deployment of lighter-weight models.

10. Related Work

RuntimeGuard-AI builds upon and differentiates itself from several active research streams.

10.1 Agentic Governance Frameworks

MI9 (Wang et al., 2025) [7]: MI9 proposes a rich “Agency-Risk Index” and FSM-based conformance. However, MI9 is purely architectural—it lacks the cryptographic binding of RuntimeGuard. RuntimeGuard could be seen as the “Enforcement Layer” for MI9’s “Policy Layer”.

GaaS (Gaurav et al., 2025) [8]: Governance-as-a-Service offers meaningful modularity but relies on centralized trust. RuntimeGuard improves on GaaS by making the audit trail decentralized and tamper-evident.

10.2 Constitutional AI and Alignment Approaches

Constitutional AI (Bai et al., 2022): Anthropic’s approach embeds safety rules into model training via RLHF. This is a *prevention* strategy—making the model itself safer. RuntimeGuard is complementary: it is a *detection and attestation* strategy that works even when the model misbehaves. Constitutional AI cannot prove post-hoc that any specific interaction was safe; RuntimeGuard can.

TrustLLM (Sun et al., 2024) [16]: TrustLLM provides comprehensive benchmarks for LLM trustworthiness across 8 dimensions. It measures; RuntimeGuard enforces. An organization could use TrustLLM to evaluate their model’s baseline safety and RuntimeGuard to prove that safety policies are continuously applied in production.

10.3 Regulatory Frameworks and Standards

NIST AI RMF (2023) [15]: The NIST AI Risk Management Framework provides a comprehensive taxonomy of AI risks and governance practices. RuntimeGuard implements the “GOVERN” and “MANAGE” functions of the RMF by providing continuous monitoring and evidence generation capabilities.

ISO/IEC 42001: The emerging AI Management System standard requires documented evidence of AI governance. RuntimeGuard’s tamper-evident logs directly satisfy the “records” requirement of such management systems.

10.4 Cryptographic Logging

Certificate Transparency (RFC 6962) [10]: Our use of Merkle Trees is directly inspired by CT. However, CT is for *static* certificates. RuntimeGuard extends CT to *dynamic* runtime events with the additional complexity of proving policy execution, not just existence.

10.48047/jocaaa.2026.35.01.87

Binary Transparency (Google): Similar to CT but for software binaries. RuntimeGuard applies the same principles to AI inference decisions.

10.5 ZKML (Zero-Knowledge Machine Learning)

ZKML [13]: Projects like EZKL and Modulus Labs focus on proving the *computational integrity* of model inference—that the model weights were applied correctly to the input. Recent work like **zkLLM** [18] demonstrates that a 13-billion parameter LLM inference can be proven in under 15 minutes. RuntimeGuard focuses on proving that the *policy logic* was correctly executed. These are complementary: a future system could use ZKML to prove the model inference and RuntimeGuard to prove the safety check. The combined system would provide complete end-to-end verifiable AI.

System	Focus	Cryptographic?	Dynamic?	Open Source?
MI9	Architecture	+	■	+
GaaS	Modularity	+	■	+
Constitutional AI	Training	+	+	+
TrustLLM	Benchmarking	+	+	■
Certificate Transparency	Static Logs	■	+	■
ZKML (EZKL)	Inference	■	+	■
RuntimeGuard-AI	Policy Enforcement	■	■	■

11. Conclusion

The EU AI Act presents a dilemma: how to mandate human oversight without breaking the speed of modern AI. We have presented **RuntimeGuard-AI**, an architecture that solves this not through policy, but through topology. By creating a distinct, asynchronous cryptographic plane for compliance, we have shown that we can have our cake (low latency) and eat it too (rigorous accountability).

Our architecture is grounded in formal proofs of Latency Separation and Tamper-Evidence. Our reference implementation proves the viability of the approach on commodity hardware. While challenges remain—specifically around the “Oracle Problem” and key management—RuntimeGuard-AI represents a concrete step toward a future where AI safety is not just a promise, but a mathematical proof.

11.1 Limitations and Future Work

The Oracle Problem: RuntimeGuard-AI proves that policies were *executed*, not that they were *correct* or *complete*. A poorly designed policy (e.g., one that misses a jailbreak vector) will be faithfully enforced but will not catch the violation. Future work should explore formal verification of natural language policies.

Trusted Setup Ceremony: Groth16 requires a one-time trusted setup. If the “toxic waste” from this ceremony is not properly destroyed, an attacker can forge proofs. We recommend using established multi-party computation (MPC) ceremonies like Zcash’s “Powers of Tau” or running a private ceremony with external auditors.

Groth16 vs Modern Alternatives: Our implementation uses Groth16 for its small proof size. However, newer systems like **Plonky2** (Polygon), **Halo2** (Zcash), and **STARKs** (StarkWare) offer transparent

10.48047/jocaaa.2026.35.01.87

setups (no toxic waste) at the cost of larger proofs. Future versions of RuntimeGuard could support these alternatives for organizations with stricter key management requirements.

Proving System	Trusted Setup?	Proof Size	Verification Time
Groth16	Yes	128 bytes	~3 ms
Plonky2	No	~45 KB	~15 ms
Halo2	No	~10 KB	~10 ms
STARKs	No	~200 KB	~50 ms

Privacy Leakage: While ZK proofs hide the *content* of logs, the *metadata* (timing, volume, batch sizes) may leak information. A sophisticated adversary could infer usage patterns. Future work should analyze differential privacy guarantees.

Scalability Ceiling: The current CPU-based implementation tops out at ~1400 proofs/second (assuming 1s proving time). For hyperscale deployments (10M+ requests/hour), GPU or FPGA acceleration is mandatory. We have designed the architecture to be hardware-agnostic, but this acceleration is not yet implemented.

12. References

- [1] European Union, “Regulation (EU) 2024/1689 of the European Parliament and of the Council,” Official Journal of the European Union, L 2024/1689, Jul. 12, 2024. <https://eur-lex.europa.eu/>
- [2] AI Act Service Desk, “EU AI Act Timeline and Key Dates,” <https://artificialintelligenceact.eu/ai-act-timeline/>
- [3] European Union, “EU AI Act Article 14: Human Oversight,” <https://artificialintelligenceact.eu/article/14/>
- [4] CEN/CENELEC, “Standardisation Request for AI Systems,” prEN 18229-1 (under development), 2025.
- [5] M. Mitchell et al., “Model Cards for Model Reporting,” FAT* 2019, pp. 220–229.
- [6] Weights and Biases, “MLOps: Continuous delivery and automation pipelines in machine learning,” Technical Report, 2023.
- [7] C. L. Wang, T. Singhal, A. Kelkar, and J. Tuo, “MI9—Agent Intelligence Protocol: Runtime Governance for Agentic AI Systems,” arXiv:2508.03858, 2025.
- [8] S. Gaurav, J. Chaudhary, et al., “Governance-as-a-Service: A Multi-Agent Framework for AI System Compliance and Policy Enforcement,” arXiv:2508.18765, 2025.
- [9] Y. Zhang et al., “NoCap: Near-Optimal ZK Acceleration via Custom Hardware,” IEEE MICRO, 2024.
- [10] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” RFC 6962, IETF, 2013. <https://www.rfc-editor.org/rfc/rfc6962>
- [11] J. Groth, “On the Size of Pairing-based Non-interactive Arguments,” EUROCRYPT 2016, pp. 305–326.

10.48047/jocaaa.2026.35.01.87

- [12] Ingonyama, “ICICLE-Snark: GPU-Accelerated ZK Proving,” Medium, 2024. <https://medium.com/@ingonyama>
- [13] D. Kang et al., “ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs,” EuroSys’24, 2024.
- [14] L. Grassi et al., “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems,” USENIX Security 2021.
- [15] NIST, “AI Risk Management Framework (AI RMF 1.0),” NIST AI 100-1, 2023.
- [16] L. Sun, Y. Huang, et al., “TrustLLM: Trustworthiness in Large Language Models,” Proc. 41st International Conference on Machine Learning (ICML), 2024. <https://arxiv.org/abs/2401.05561>
- [17] Y. Bai, S. Kadavath, et al., “Constitutional AI: Harmlessness from AI Feedback,” arXiv:2212.08073, Dec. 2022. <https://arxiv.org/abs/2212.08073>
- [18] H. Sun, J. Zhu, et al., “zkLLM: Zero Knowledge Proofs for Large Language Models,” arXiv:2404.16109, Apr. 2024. <https://arxiv.org/abs/2404.16109>