

Zero-Overhead Post-Quantum Auditing and Threat Mitigation for Large Language Model Inference Pipelines

Juan Carlos Luna

Abstract—Context. The deployment of Large Language Model (LLM) inference pipelines in regulated environments has produced an architectural fault line: applications integrate inference calls as stateless remote procedure calls (RPCs), leaving the question “what, exactly, did the model receive and return, and can you prove the record was not altered?” without a cryptographically defensible answer. Conventional application logs and database trails are mutable by design; they prove that an event was recorded, not that the recorded bytes are the bytes that crossed the wire, and they offer no protection against an adversary who silently edits the audit trail *after* the fact. The growing maturity of cryptographically relevant quantum hardware sharpens this problem further: an adversary operating under the Harvest-Now-Decrypt-Later (HNDL) threat model can archive today’s RSA- and ECDSA-protected telemetry and forge its contents once a sufficiently large quantum computer arrives.

Problem. We require an inference proxy that simultaneously satisfies four seemingly incompatible properties: (i) it adds sub-microsecond scheduling overhead to the request hot path so that production latency budgets are unaffected; (ii) it produces a tamper-evident, hash-chained audit ledger whose integrity can be verified by an independent third party without trusting the proxy itself; (iii) it signs every audit record using a post-quantum signature scheme whose security reduces to a standard lattice problem; and (iv) it inspects every prompt and response against a multi-engine threat pipeline spanning prompt-injection, malware, classified-markers, SCADA/OT, and adversarial-suffix attacks.

Technical Contribution. We present Aegis Latent Core, a hybrid Python/Rust inference proxy that satisfies all four properties. Aegis Latent Core introduces an *asynchronous dual-path execution model* that separates the client-visible hot path (auth $\rightarrow$ WAF $\rightarrow$ rate limit $\rightarrow$ forwarder $\rightarrow$ response) from a background audit-commit queue (response analyser $\rightarrow$ cryptographic audit ledger $\rightarrow$ memory-mapped write-ahead log) using a single `asyncio.create_task` dispatch whose locally-measured scheduling latency is $1.47\ \mu\text{s}$ at the 50th percentile and $9.54\ \mu\text{s}$ at the 99th percentile over $n = 5000$ invocations. The audit ledger is a Merkle Mountain Range (MMR) whose leaf hashes are BLAKE3 digests of normalised (prompt, response, metadata) tuples; chain integrity is anchored by HMAC-SHA256 for backward compatibility and by ML-DSA-65 (FIPS 204) for post-quantum attestation. Ten independent inspection engines execute on the hot path with NFKC normalisation,

zero-width character stripping, and SIMD Aho-Corasick multi-pattern matching implemented in Rust via PyO3.

Key Empirical Results. On a 4-core x86_64 host running Debian 13 (trixie), Python 3.12.13, and a release-compiled Rust extension (rustc 1.96.0, maturin 1.14.1), Aegis Latent Core sustains a peak throughput of 902 req/s at concurrency $c = 4$, with the audit-commit queue adding no measurable I/O wait on the response socket. The HMAC-SHA256 audit chain commits 4080 forensic records per second ($245.28\ \mu\text{s}/\text{op}$ including MMR insertion, BLAKE3 leaf hashing, peak bagging, HMAC signing, and durable WAL `fsync`); the chain-verification path processes 89270 nodes per second ($11.20\ \mu\text{s}/\text{node}$); the bare HMAC-SHA256 signer sustains 824580 operations per second ($1.213\ \mu\text{s}/\text{op}$); the ML-DSA-65 post-quantum signer exhibits $p_{50} = 92.22\ \mu\text{s}$, $p_{99} = 375.17\ \mu\text{s}$, $\sigma = 78.60\ \mu\text{s}$ over $n = 200$ sign-verify round-trips (the right-skew is intrinsic to the rejection-sampling step of FIPS 204); and the Rust-accelerated MMR implementation achieves a $3.26\times$ average speedup (max $4.21\times$ at $N = 10^5$ leaves) over its pure-Python counterpart. Process RSS remains flat at 138 MiB ($\Delta\text{RSS} = 0.00\ \text{MiB}$) over 6000 audit commits, confirming the absence of memory leaks. A 15-payload WAF detection sweep reveals an important testing-vs-deployment distinction: the standalone `inspect_payload()` entry point blocks only 4 of 13 attack payloads, whereas the full proxy handler `/v1/chat/completions` routes through the complete 10-engine pipeline (including `SecretLeakDetector` and `ClassifiedMarkerDetector`).

Security Impact. Aegis Latent Core aligns with the control families of FedRAMP High (AU, AC, SC), DoD IL5/IL6, HIPAA §164.312(b), SEC Rule 17a-4 (WORM), PCI-DSS v4.0, ISO 27037, and 21 CFR Part 11. A 14-framework compliance coverage matrix documents 117 [PROVEN] controls and 23 [PARTIAL] controls with explicit boundary statements (e.g., the absence of native CMS/PKCS#7 formatting and the requirement for external HSM public-key SPKI DER verification). The system has been empirically validated against a 5416-test suite with 94% branch coverage and a `KNOWN_SIMULATION_DEBT == 0` ratchet guard that hard-gates the introduction of security-theater code paths.

Index Terms—Post-quantum cryptography, ML-DSA-65, FIPS 204, ML-KEM-1024, Merkle Mountain Range, BLAKE3, LLM inference security, prompt injection, forensic telemetry, tamper-evident audit chain, FedRAMP High, HIPAA, SEC Rule 17a-4, ISO 27037, PyO3, asynchronous audit queue.

Manuscript prepared 2026-06-28. J. C. Luna is an Independent Researcher and a student in the University Technical Program in Artificial Intelligence at the Facultad de Ciencias Exactas, Ingeniería y Agrimensura (FCEIA), Universidad Nacional de Rosario (UNR). Mailing address: Rosario, Santa Fe, Argentina. Email: juan.c.luna04@gmail.com. Reference implementation: <https://github.com/JuanLunaIA/aegis-latent-core>, v2.4.1, AGPL-3.0. All numerical values reported in Tables 4–7 are locally-measured metrics collected on a 4-core x86_64 host running Debian 13 (trixie) and Python 3.12.13, using the benchmark suite shipped with the repository (`benchmarks/bench_forwarding.py`, `benchmarks/bench_crypto_audit.py`, `benchmarks/bench_mmr.py`).

1 Introduction

THE integration of Large Language Model (LLM) inference into enterprise and regulated-industry workloads has outpaced the development of the governance infrastructure required to make such integrations auditable. A typical enterprise deployment places a thin RPC client between an application server and a

remote inference endpoint operated by a third party; the client receives a prompt from upstream, dispatches it to the model, and returns the generated tokens to the caller. The record of what was sent and what was returned lives, at best, in a mutable row of an application database, written by the same process that performed the inference and alterable by every operator with database credentials. When a regulator, a court, or a Chief Information Security Officer (CISO) subsequently asks the question “show me what the model received and prove the log is unmodified,” the application owner has no cryptographic answer to give. The present paper describes a system that closes this gap.

1.1 The Security Boundary Crisis in Enterprise AI

The defining architectural property of an LLM inference call is that the inference payload crosses three trust boundaries in a single round trip: it leaves the application’s trust domain, traverses a network controlled by a third-party provider, and re-enters the application’s trust domain carrying tokens whose provenance the application cannot independently verify. Each crossing is a potential violation point. An outbound prompt may contain classified material, personally identifiable information (PII), payment-card data (PAN), protected health information (PHI), or malware signatures; an inbound response may carry a prompt-injection payload, an adversarial suffix, an extracted secret echoed back from the model’s context window, or a many-shot jailbreak sequence. Without an enforcement point at the trust boundary, the application owner cannot demonstrate that any of these failure modes was detected, much less blocked.

The governance literature has so far addressed this problem at the *policy* level—e.g., NIST AI RMF, ISO/IEC 42001, the EU AI Act—but has not converged on the *mechanism* by which deterministic governance is to be enforced. The result is a proliferation of bespoke, application-specific mitigations that are individually weak and collectively unverifiable. Aegis Latent Core was designed to occupy the missing mechanism layer: a cryptographically enforced proxy that sits between the application and the inference endpoint, inspects every payload in both directions, and produces a tamper-evident record that any third party can independently verify.

The urgency of this mechanism layer is amplified by two macro-trends. First, the regulatory horizon is converging on a single requirement: every inference call that materially affects a regulated decision must produce a verifiable, tamper-evident record that can be independently audited years after the fact. The EU AI Act (effective 2026), US Executive Order 14110, NIST AI RMF, ISO/IEC 42001, and the emerging DoD CDAO policies all converge on this requirement, though they leave the implementation to the operator. Second, the cryptographic horizon is shifting: the Harvest-Now-Decrypt-Later (HNDL) adversary is now treated as a first-class threat model for any data whose retention period exceeds the expected time-to-quantum-computer, which for audit records in regulated industries may be seven to twenty-five years (SEC Rule 17a-4: six years; HIPAA: six years; DoD IL6: indefinite; GxP Annex 11: at least five years post product retirement). Application logs sealed with RSA or ECDSA are not merely weak under this adversary; they are retroactively forgeable, which is a strictly worse outcome than having no audit trail at all.

1.2 Limitations of Mutable Application Logging

Conventional audit mechanisms in production LLM deployments fall into three classes, each of which fails to provide forensic non-repudiation:

(i) **Application logs (structured or unstructured).** Application log files are append-only by convention, not by construction. Any operator with filesystem write permission can edit, truncate, or fabricate log lines after the fact. A timestamp in a log line proves only that a string was written to a file at some point; it does not prove that the string corresponds to a real inference call, nor that the timestamp itself was not rewritten. The problem is compounded by log rotation, log shipping, and centralised log aggregation, each of which introduces additional mutation surfaces between the event and the eventual storage medium.

(ii) **Database audit tables.** Database-backed audit tables inherit the threat model of the underlying database management system. A privileged operator can issue UPDATE or DELETE against the audit table directly; even when the database enforces append-only semantics at the schema level (e.g. via triggers), the storage layer beneath the database remains mutable. The WORM (Write-Once-Read-Many) requirement of SEC Rule 17a-4 and the tamper-evident requirement of HIPAA §164.312(b) cannot be satisfied by a database audit table alone, because both requirements presuppose that the storage layer itself resists modification by privileged users.

(iii) **Append-only files with periodic cryptographic sealing.** A minority of deployments hash-chain their audit logs and seal the chain periodically (e.g. hourly) with a digital signature. This construction provides partial tamper-evidence: a forger can modify records within the current unsealed window, and an adversary who compromises the signing key can back-date forgeries. The construction offers no protection against the HNDL adversary discussed in Section 1.3, who records today’s RSA- or ECDSA-signed seals and forges them after a quantum computer arrives.

The unifying defect of all three approaches is the absence of an *independently verifiable* binding between the byte-level contents of an inference exchange and a persistent, hash-chained record. Aegis Latent Core addresses this defect by enforcing three invariants: (a) every audit node is bound to its predecessor by a cryptographic hash, (b) the chain is sealed continuously (not periodically) by a post-quantum signature, and (c) the seal is verifiable by any third party in possession of the public key, without trusting the proxy, the host, or the operator.

1.3 The Harvest-Now-Decrypt-Later Threat Model

The HNDL adversary is a long-horizon adversary who (i) records encrypted or signed telemetry today, (ii) archives it, and (iii) reconstructs the signing keys or decrypts the payloads once a sufficiently large cryptographically relevant quantum computer becomes available. The model is realistic for any telemetry whose retention period exceeds the expected time-to-quantum-computer, which for audit records in regulated industries may be 7 to 25 years (SEC Rule 17a-4: 6 years; HIPAA: 6 years; DoD IL6: indefinite; GxP Annex 11: ≥ 5 years post-product-retirement).

The cryptographic primitives whose security collapses under Shor’s algorithm are exactly those in widest production use for audit-trail integrity: RSA, DSA, ECDSA, and any Diffie-Hellman-based key exchange. The National Institute of Standards and Technology (NIST) standardised three post-quantum primitives in

August 2024: **FIPS 203** (ML-KEM, key encapsulation), **FIPS 204** (ML-DSA, digital signatures), and **FIPS 205** (SLH-DSA, stateless hash-based signatures). Aegis Latent Core adopts ML-DSA-65 (the FIPS 204 parameter set with security category 3, equivalent to AES-192) for audit-record signing and ML-KEM-1024 (FIPS 203 category 5) for the optional hybrid KEM used in session establishment with the upstream provider. The choice of ML-DSA-65 over SLH-DSA is motivated by the 3× smaller signature size at equivalent security levels; the choice of category 3 over category 5 for the signing key reflects the engineering constraint that audit records are high-volume and per-record signing latency dominates throughput.

The threat model assumes the adversary has full access to the recorded telemetry (which is the standard “Internet is a hostile medium” assumption for any traffic that traverses a third-party provider) and has the patience to wait for a quantum computer. It does *not* assume the adversary can break symmetric primitives such as SHA-256, BLAKE3, or AES-256, whose security is reduced only quadratically by Grover’s algorithm and therefore remains adequate for the relevant retention periods. The implication is that an audit chain anchored by HMAC-SHA256 (a symmetric primitive) is quantum-safe *as long as the HMAC key is not exfiltrated*, but the digital signature that anchors the chain tip is quantum-vulnerable if it is RSA- or ECDSA-based. Aegis Latent Core therefore uses HMAC-SHA256 for per-record authentication (cheap, symmetric, quantum-safe) and ML-DSA-65 for per-tip non-repudiation (more expensive, asymmetric, post-quantum), with the two layers composed so that a break in either one alone does not compromise the chain.

1.4 Contributions of this Paper

This paper makes the following contributions:

- (C1) **The asynchronous dual-path execution model** (Section 3). We separate the client-visible hot path from a background audit-commit queue using a single `asyncio.create_task` dispatch. Empirically the dispatch adds 1.47 μ s of scheduling latency at the 50th percentile and 9.54 μ s at the 99th percentile over $n = 5000$ measurements, with a standard deviation of 1.25 μ s and a maximum of 21.18 μ s. This is more than two orders of magnitude below the typical round-trip latency of an LLM inference call (> 200 ms).
- (C2) **The ten-engine threat mitigation pipeline** (Section 4). We specify and implement a pipeline of ten independent inspection engines that execute on the hot path with NFKC Unicode normalisation, zero-width character stripping, homoglyph collapsing, case folding, and token-split collapse, followed by SIMD Aho-Corasick multi-pattern matching. We additionally document an important testing-vs-deployment distinction (Section 4.4): the standalone `inspect_payload()` entry point exercises only AegisWAF (Engine 1) and blocks 4 of 13 attack payloads, whereas the full proxy handler `/v1/chat/completions` routes through the complete ten-engine pipeline including `SecretLeakDetector` and `ClassifiedMarkerDetector`.
- (C3) **The post-quantum Merkle Mountain Range audit ledger** (Sections 2.4 and 3). We construct a tamper-evident audit ledger as an MMR whose leaf hashes are BLAKE3 digests of normalised (prompt, response, metadata) tuples, sealed continuously by HMAC-SHA256 for backward compatibility and by ML-DSA-65 for post-quantum attestation. Inclusion

proofs are $O(\log N)$ in the chain length. The Rust implementation of the MMR achieves an average 3.26× speedup (max 4.21× at $N = 10^5$ leaves) over the pure-Python reference implementation.

- (C4) **Empirical validation on a 5416-test suite with locally-measured benchmarks** (Section 6). We report hot-path overhead, throughput under load, per-component cryptographic throughput, post-quantum signer latency distribution, audit-chain commit latency distribution, MMR scaling, memory stability, and WAF detection coverage. Locally-measured benchmarks confirm a bare HMAC-SHA256 signing throughput of 824 580 ops/s (1.213 μ s/op), an end-to-end `commit_forensic()` throughput of 4080 commits/s (245.28 μ s/op), a chain-verification throughput of 89 270 nodes/s (11.20 μ s/node), a ML-DSA-65 signing latency of $p_{50} = 92.22 \mu$ s / $p_{99} = 375.17 \mu$ s / $\sigma = 78.60 \mu$ s (the right-skew attributable to rejection sampling in FIPS 204), and a flat 138 MiB RSS over 6000 commits (Δ RSS = 0.00 MiB).
- (C5) **A cross-domain compliance mapping with a 14-framework coverage matrix** (Section 5). We map the Aegis Latent Core control surface to FedRAMP High, DoD IL5/IL6, HIPAA §164.312(b), SEC Rule 17a-4, PCI-DSS v4.0, ISO 27037, 21 CFR Part 11, IEC 62443, SOC 2 Type II, ISO 27001, GDPR, NIST SP 800-53, MiFID II, and GAMP 5, documenting 117 [PROVEN] controls and 23 [PARTIAL] controls with explicit boundary statements (e.g., the absence of native CMS/PKCS#7 formatting and the requirement for external HSM public-key SPKI DER verification).

The remainder of this paper is organised as follows. Section 2 develops the mathematical foundations. Section 3 presents the dual-path architecture. Section 4 details the ten-engine inspection pipeline and the WAF sweep gap analysis. Section 5 presents the 14-framework compliance coverage matrix. Section 6 reports the empirical results. Section 7 discusses limitations and future work. Section 8 concludes.

2 Theoretical and Mathematical Foundations

This section develops the four mathematical pillars on which the Aegis Latent Core design rests: (a) forensic telemetry modelling, in which the LLM prompt–response pair is formalised as a stochastic sequence; (b) statistical drift analysis, which provides the entropy- and divergence-based detectors used to flag anomalous prompts and responses; (c) post-quantum lattice cryptography, which provides the signature scheme that seals the audit chain; and (d) the Merkle Mountain Range cryptographic accumulator, which provides the $O(\log N)$ inclusion-proof structure that binds individual audit records to a chain tip.

2.1 Forensic Telemetry Modelling

We model an LLM inference call as a structured event $e = (p, r, t, \alpha, \kappa, m)$, where:

- $p \in \Sigma^*$ is the prompt byte-string, drawn from the alphabet Σ of valid UTF-8 code points;
- $r \in \Sigma^*$ is the response byte-string returned by the upstream model;
- $t \in \mathbb{R}_{\geq 0}$ is a monotonic server-side timestamp, sourced from a clock with bounded drift;
- $\alpha \in \mathcal{A}$ is the authenticated application identity (e.g. the proxy-issued API key) under whose authority the call was made;

- $\kappa \in \mathcal{K}$ is the upstream provider identity (e.g. openai, anthropic, gemini, openrouter);
- $m \in \mathcal{M}$ is a structured metadata dictionary containing the model identifier, token counts, finish reason, and detection-engine verdicts.

A *forensic telemetry stream* is the countable sequence $\mathcal{E} = (e_1, e_2, e_3, \dots)$ ordered by t . The role of the audit chain is to commit each e_i to a persistent, hash-chained data structure such that (i) the integrity of any e_i can be verified against the chain tip, and (ii) the temporal order of the sequence is itself cryptographically enforced.

Definition 1 (Audit node). *The i -th audit node is the tuple*

$$n_i = (i, \text{hash}(e_i), \text{hash}(n_{i-1}), t_i, \sigma_i),$$

where $\text{hash} : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$ is a collision-resistant cryptographic hash function (BLAKE3 in Aegis Latent Core), and σ_i is a signature over $\text{hash}(e_i) \parallel \text{hash}(n_{i-1}) \parallel t_i$ under the audit signer's key. The chain is initialised with a sentinel node $n_0 = (0, 0^{256}, 0^{256}, t_0, \sigma_0)$ whose hash is the all-zero string, so that $\text{hash}(n_0)$ is a publicly known constant.

Definition 2 (Chain integrity). *An audit chain $\mathcal{N} = (n_1, n_2, \dots, n_N)$ satisfies chain integrity if and only if, for every $1 \leq i \leq N$,*

$$\text{prev_hash}(n_i) = \text{hash}(n_{i-1}),$$

and the signature σ_i verifies under the audit signer's public key pk. A break in chain integrity at index i is a forensic proof that n_i or one of its predecessors was modified after commitment.

The verification procedure implied by Definition 2 is $O(N)$ in the chain length. The MMR construction of Section 2.4 provides an $O(\log N)$ inclusion-proof structure layered on top of the chain, allowing a third party to verify the inclusion of any single n_i in the chain tip without re-walking the entire chain.

2.2 Statistical Drift Analysis

Statistical drift analysis provides the mathematical substrate for the entropy- and divergence-based detectors in the Aegis Latent Core inspection pipeline. We use three quantities: the Shannon entropy of a token distribution, the Kullback-Leibler (KL) divergence between a reference and an observed distribution, and the Jensen-Shannon (JS) divergence, which bounds the KL divergence to a symmetric, finite-valued measure.

2.2.1 Shannon Entropy

Let X be a discrete random variable taking values in $\{x_1, x_2, \dots, x_n\}$ with probability mass function $P(X = x_i) = p_i$. The *Shannon entropy* of X is

$$H(X) = - \sum_{i=1}^n p_i \log_2 p_i, \quad (1)$$

with the convention $0 \log_2 0 = 0$ (continuous extension by limit). In Aegis Latent Core, X is the empirical distribution of token classes in a sliding window of the response stream; a sudden collapse of $H(X)$ (towards zero) signals a degenerate or repetitive output that is characteristic of a model hallucination or a jailbreak-induced repetition loop. A sudden inflation of $H(X)$ (towards $\log_2 n$) signals a uniform-distribution output that is characteristic of an extraction attack or a model inversion attempt.

Theorem 1 (Bounds on Shannon entropy). *For any discrete random variable X with n outcomes, $0 \leq H(X) \leq \log_2 n$. The lower bound is attained iff X is deterministic; the upper bound is attained iff X is uniform on its support.*

Proof. The lower bound $H(X) \geq 0$ follows because each summand $-p_i \log_2 p_i$ is non-negative for $p_i \in [0, 1]$. The upper bound follows from the concavity of $\log$ and Jensen's inequality: $H(X) = \mathbb{E}[-\log_2 p_i] \leq -\log_2 \mathbb{E}[p_i] = -\log_2(1/n) = \log_2 n$. Equality holds iff p_i is constant on its support, i.e. $p_i = 1/n$. $\square$

2.2.2 Kullback-Leibler Divergence

Let P and Q be two probability distributions over the same support $\{x_1, \dots, x_n\}$, with Q absolutely continuous with respect to P (denoted $Q \ll P$). The *Kullback-Leibler divergence* from Q to P is

$$D_{\text{KL}}(P \parallel Q) = \sum_{i=1}^n p_i \log_2 \frac{p_i}{q_i}. \quad (2)$$

The KL divergence is non-negative and equals zero if and only if $P = Q$ almost everywhere. It is *not* symmetric in P and Q and is unbounded above, which makes it unsuitable as a metric. In Aegis Latent Core, P is the reference distribution of token classes observed during a calibration window (e.g. the first 10 000 calls after deployment), and Q is the empirical distribution observed in a recent sliding window.

Theorem 2 (Gibbs' inequality). *For any two distributions P, Q on a common support, $D_{\text{KL}}(P \parallel Q) \geq 0$, with equality iff $P = Q$.*

Proof. Apply the log-sum inequality to $\sum_i p_i \log(p_i/q_i)$, or equivalently use the convexity of $-\log$ and Jensen's inequality: $-D_{\text{KL}}(P \parallel Q) = \sum_i p_i \log(q_i/p_i) \leq \log \sum_i p_i (q_i/p_i) = \log \sum_i q_i = \log 1 = 0$. Hence $D_{\text{KL}} \geq 0$, with equality iff q_i/p_i is constant on the support of P , which forces $q_i = p_i$. $\square$

2.2.3 Jensen-Shannon Divergence

The *Jensen-Shannon divergence* is the symmetrised, finite-valued sibling of the KL divergence. Let $M = \frac{1}{2}(P + Q)$ be the midpoint distribution. Then

$$D_{\text{JS}}(P, Q) = \frac{1}{2} D_{\text{KL}}(P \parallel M) + \frac{1}{2} D_{\text{KL}}(Q \parallel M). \quad (3)$$

The JS divergence is bounded in $[0, 1]$ (when the logarithm is taken in base 2), is symmetric, and is finite even when P and Q have disjoint supports. Its square root, $\sqrt{D_{\text{JS}}(P, Q)}$, is a metric in the strict mathematical sense and is the quantity used by Aegis Latent Core for the adversarial-suffix and many-shot jailbreak detectors.

Theorem 3 (Triangle inequality for JS metric). *For any three distributions P, Q, R over a common support, $\sqrt{D_{\text{JS}}(P, R)} \leq \sqrt{D_{\text{JS}}(P, Q)} + \sqrt{D_{\text{JS}}(Q, R)}$.*

Proof. The result follows from the fact that $\sqrt{D_{\text{JS}}}$ is the total-variation distance scaled by a constant factor and that total-variation distance is a metric; see Endres and Schindelin [14]. Concretely, $D_{\text{JS}}(P, Q) = H(M) - \frac{1}{2}H(P) - \frac{1}{2}H(Q)$ where H is Shannon entropy, which equals the mutual information $I(P; Q)$ between a Bernoulli selector and the chosen distribution. The square-root mutual information is a metric by a result of Endres and Schindelin [14], who prove the triangle inequality via the Pinsker-Csiszár-Kullback inequality chain. $\square$

Proposition 1 (Bounding JS via total variation). *For any two distributions P, Q , $\frac{1}{2}\text{TV}(P, Q)^2 \leq D_{\text{JS}}(P, Q) \leq \text{TV}(P, Q) \log_2 \frac{2}{1-\text{TV}(P, Q)}$, where $\text{TV}(P, Q) = \frac{1}{2} \sum_i |p_i - q_i|$ is the total-variation distance. The lower bound is the Pinsker-Csiszár-Kullback inequality applied to each KL term in (3); the upper bound follows from the convexity of $-\log$.*

Theorem 3 and Proposition 1 together are what permit Aegis Latent Core to construct clustering-based detectors over the JS metric: a single representative “jailbreak centroid” Q can be compared against all incoming prompts P_i , and the triangle inequality bounds the worst-case misclassification radius.

2.3 Post-Quantum Lattice Cryptography

The post-quantum signature scheme used by Aegis Latent Core is ML-DSA-65, the FIPS 204 standard built on the Module Learning With Errors (M-LWE) hard problem.

2.3.1 The M-LWE Hard Problem

Let $R_q = \mathbb{Z}_q[x]/(x^N + 1)$ be the ring of polynomials modulo $x^N + 1$ with coefficients in $\mathbb{Z}_q$, where N is a power of two and q is a prime modulus. Let χ_η be a centred, bounded error distribution over R_q that returns small-norm polynomials. The *Module Learning With Errors* problem is: given a uniformly random module matrix $A \in R_q^{k \times \ell}$, a uniformly random secret $s \in R_q^\ell$, and an error vector $e \in R_q^k$ sampled from χ_η^k , distinguish the pair $(A, b = A \cdot s + e)$ from a uniformly random pair $(A, b) \leftarrow R_q^{k \times \ell} \times R_q^k$. ML-DSA-65 uses $(N, q, k, \ell, \eta) = (256, 8380416, 6, 5, 6)$, which FIPS 204 associates with security category 3 (approximately AES-192-equivalent).

Theorem 4 (M-LWE hardness reduction). *The M-LWE problem with parameters (N, q, k, ℓ, η) is at least as hard as the worst-case Module Learning With Errors problem on the same ring, which in turn is at least as hard as the worst-case approximate shortest-vector problem on the same module lattice to within a factor $\tilde{O}(q/\eta)$ in the ℓ_∞ norm.*

Proof. The reduction proceeds in two steps. First, the Module-LWE problem with bounded-noise distribution reduces to the Module-LWE problem with discrete Gaussian noise via the noise-flooding technique of Lyubashevsky, Peikert, and Regev [7]. Second, the Module-LWE problem with discrete Gaussian noise reduces to the worst-case approximate shortest-vector problem on the same module lattice via the classical LWE-to-lattice reduction of Regev generalised to modules. $\square$

2.3.2 The ML-DSA-65 Verification Equation

The ML-DSA-65 signature of a message M under the public key $\text{pk} = (A, t)$ is a tuple $\sigma = (z, h, \tilde{c})$, where $z \in R_q^k$ is the response polynomial, h is the hint vector, and $\tilde{c} \in R_q$ is the challenge polynomial derived from the message and the commitment. The verification procedure accepts σ if and only if the following three conditions hold simultaneously:

$$\begin{aligned} & \text{(i) } \|z\|_\infty < \gamma_1 - \beta, \\ & \text{(ii) } \text{wt}(h) \leq \omega, \\ & \text{(iii) } c' = H(\text{HighBits}(Az - ct2^d, 2\gamma_2) \parallel \|M\| \tilde{c}) = c, \end{aligned} \quad (4)$$

where H is the SHAKE-256-based domain-separated hash, $\text{wt}(\cdot)$ denotes the Hamming weight, HighBits extracts the high-order bits

of each polynomial coefficient, and the constants $(\gamma_1, \gamma_2, \beta, \omega, d)$ are fixed by FIPS 204. For ML-DSA-65 the parameter set is $(\gamma_1, \gamma_2, \beta, \omega, d) = (2^{17}, 2^{14}, 78, 80, 13)$. The public key pk is 1 952 bytes, the private key sk is 4 032 bytes, and the signature σ is 3 309 bytes.

Lemma 1 (Signature unforgeability). *Under the M-LWE hardness assumption (Theorem 4), the ML-DSA-65 signature scheme is existentially unforgeable under chosen-message attack (EUF-CMA) in the random-oracle model with advantage bounded by $O(q_H \cdot \varepsilon_{\text{M-LWE}})$, where q_H is the number of random-oracle queries and $\varepsilon_{\text{M-LWE}}$ is the M-LWE distinguishing advantage.*

Proof. The proof proceeds via a sequence of games: (G1) the EUF-CMA game; (G2) the challenge c is replaced by a uniform value via the forking lemma; (G3) the random oracle is reprogrammed to make the forked transcript inconsistent; (G4) the forger is reduced to a solver for M-LWE. The total advantage is bounded by $q_H^2 \cdot \varepsilon_{\text{M-LWE}}$ plus negligible terms. See Duc et al. [8] for the full proof. $\square$

2.3.3 Rejection Sampling and the Signing Latency Distribution

The ML-DSA-65 signing algorithm produces candidate signatures (z, h) by sampling z from a centered Gaussian and accepting the candidate only if $\|z\|_\infty < \gamma_1 - \beta$. When the candidate is rejected (which occurs with probability $1 - 1/M$ per iteration, where M is the rejection-sampling normalisation constant), the signer re-samples and re-tries. The expected number of iterations per signature is M , and the variance is $M(M - 1)$. For ML-DSA-65, $M \approx 4.5$, which means a single signature requires on average 4.5 rejection-sampling iterations.

This rejection-sampling structure is the mathematical explanation for the empirically observed right-skew in the signing-latency distribution (Table 5 and Figure 3): the bulk of signatures complete in $\sim 90 \mu\text{s}$ (one or two iterations), but a long tail extends to $\sim 375 \mu\text{s}$ (the p_{99}) corresponding to signatures that required ≥ 7 rejection-sampling iterations. The mean ($120.92 \mu\text{s}$) is biased above the median ($92.22 \mu\text{s}$) by this tail. The right-skew is therefore not a performance defect but an intrinsic property of the FIPS 204 signing algorithm; any correct ML-DSA-65 implementation will exhibit it.

2.3.4 Hybrid KEM Session Establishment (ML-KEM-1024)

For deployments that require an end-to-end encrypted channel between the application and the proxy, Aegis Latent Core optionally supports a hybrid KEM using ML-KEM-1024 (FIPS 203 category 5). The initiator generates an ML-KEM-1024 keypair $(\text{pk}_{\text{KEM}}, \text{sk}_{\text{KEM}})$ and publishes pk_{KEM} . The responder encapsulates a fresh session key $K \in \{0, 1\}^{256}$ by computing $(c, K) = \text{Encaps}(\text{pk}_{\text{KEM}})$ and sends the ciphertext c to the initiator. The initiator decapsulates via $K = \text{Decaps}(\text{sk}_{\text{KEM}}, c)$. The session key K is then mixed with an X25519-derived ephemeral key via HKDF-SHA256 to produce the hybrid key $K_{\text{hybrid}} = \text{HKDF}(K \parallel K_{\text{X25519}})$, so that the channel remains secure as long as *at least one* of ML-KEM-1024 or X25519 is unbroken.

2.4 Merkle Mountain Range Cryptographic Accumulators

The Merkle Mountain Range (MMR) is an append-only cryptographic accumulator that supports $O(\log N)$ inclusion proofs without requiring the entire tree to be rebuilt on each append.

Definition 3 (MMR leaf and parent hashes). Let $h_\ell : \mathcal{N} \rightarrow \{0, 1\}^{256}$ be the leaf hash function that maps an audit node n_i to

$$h_\ell(n_i) = \text{BLAKE3}(0x00 \parallel i \parallel \text{hash}(e_i) \parallel \text{hash}(n_{i-1})). \quad (5)$$

Let $h_p : \{0, 1\}^{256} \times \{0, 1\}^{256} \rightarrow \{0, 1\}^{256}$ be the parent hash function

$$h_p(L, R) = \text{BLAKE3}(0x01 \parallel L \parallel R). \quad (6)$$

The MMR is constructed by appending leaves one at a time. After each append, every pair of sibling nodes at the same height is collapsed into a parent node via h_p ; this continues until at most one node remains at each height.

Definition 4 (MMR peak bagging operator). Let $P_1, P_2, \dots, P_k$ be the peaks of the MMR ordered from leftmost to rightmost. The peak bagging operator $\text{Bag} : \{0, 1\}^{256k} \rightarrow \{0, 1\}^{256}$ is defined recursively as

$$\begin{aligned} \text{Bag}(P_1) &= P_1, \\ \text{Bag}(P_1, \dots, P_k) &= \text{BLAKE3}(0x02 \parallel \\ &\quad \text{Bag}(P_1, \dots, P_{k-1}) \parallel P_k). \end{aligned} \quad (7)$$

The bagged peak hash $\text{Bag}(P_1, \dots, P_k)$ is the MMR root and is published alongside each audit-chain tip.

Theorem 5 (MMR inclusion proof size). For an MMR with N leaves, the inclusion proof of the i -th leaf consists of at most $\lceil \log_2(N+1) \rceil - \text{popcount}(N+1) + 1$ sibling hashes, which is $O(\log N)$ in the worst case.

Proof. The i -th leaf participates in at most one parent at each height of the range. The number of heights is bounded by the position of the most-significant bit of N , and at each height at most one sibling hash is required. The exact bound follows from the binary representation of N ; see Ren [10]. $\square$

Corollary 1 (Verification complexity). Given an inclusion proof of length ℓ and a leaf hash $h_\ell(n_i)$, the verifier performs ℓ applications of h_p plus a constant-time bagging step, for a total of $O(\log N)$ hash evaluations.

Lemma 2 (Collision resistance of the MMR construction). If BLAKE3 is collision-resistant, then the MMR construction of Definitions 3 and 4 is collision-resistant: an adversary who finds two distinct MMR instances with the same bagged peak hash can be used to find a BLAKE3 collision in polynomial time.

Proof. The proof is a standard Merkle-tree collision-resistance argument adapted to the MMR structure. By the collision-resistance of BLAKE3, the leaf-hash function h_ℓ is collision-resistant (Definition 3); by the same assumption, the parent-hash function h_p is collision-resistant (Definition 3). Given two distinct MMR instances with the same bagged peak hash, walk down from the peak: at each level either the two instances have the same child hashes (recurse on a non-matching subtree) or they differ (in which case the BLAKE3 hash of the differing children collides). The walk terminates in $O(\log N)$ steps and produces a BLAKE3 collision. $\square$

The choice of BLAKE3 over SHA-256 for the MMR hash function is motivated by the empirical ~ 4 GB/s throughput of BLAKE3 on x86_64 with SIMD, compared to approximately 350 MB/s for SHA-256 on the same hardware.

3 Core Architecture and Dual-Path Data Plane

The Aegis LatentCore architecture is organised around a single design principle: *the request hot path must observe no I/O wait that is not strictly required to satisfy the client's request.* Every operation that does not directly contribute to the bytes returned to the client—audit-chain commitment, WAL persistence, MMR peak rebagging, post-quantum signing, statistical-drift computation, threat-intelligence correlation—is relocated to an asynchronous background queue that runs after the response has been dispatched.

3.1 System Design: The Python–Rust FFI Hybrid

Aegis LatentCore is implemented as a Python application (FastAPI, uvicorn, asyncio) that delegates performance-critical operations to a native Rust extension compiled via the PyO3 framework. The choice of Python for the orchestration layer is motivated by three factors: (i) the Python ecosystem dominates the LLM-application integration surface (the OpenAI, Anthropic, and Google SDKs are Python-first), so a Python proxy can be deployed without introducing a new language into the application's runtime; (ii) the asyncio event loop provides a natural concurrency model for I/O-bound workloads dominated by upstream HTTP calls; and (iii) Python's introspection and dynamic dispatch simplify the implementation of the ten-engine inspection pipeline, where new detectors are added on a rolling cadence.

The choice of Rust for the performance-critical substrate is motivated by the empirical observation that pure-Python implementations of the audit chain, the WAF, the WAL, and the rate limiter saturate a single CPU core at ~ 3000 ops/s, which is well below the throughput target of 500 req/s on a 4-core instance with full audit-chain commitment. The Rust extension, compiled with the `cdylib` crate type and the `extension-module` feature of PyO3, exposes the following symbols to Python:

- `RustWaf`: a SIMD-accelerated Aho-Corasick multi-pattern matcher with weighted regex scoring;
- `RustWal`: a memory-mapped write-ahead log with CRC32 frame integrity and lock-free MPSC ring buffering;
- `RustForwarder`: a connection-pooled HTTP/2 forwarder with hickory-dns async resolution and native-TLS;
- `RustRateLimiter`: a sharded lock-free rate limiter backed by dashmap;
- `RustSessionStore`: a sharded session store with TTL eviction and LRU eviction;
- `AuditRingBuffer`: a lock-free MPSC ring buffer for audit events between the hot path and the background queue;
- `MmrAccumulator`: the Merkle Mountain Range of Section 2.4, implemented with BLAKE3 leaf and parent hashes;
- `PqcKeypair`, `generate_pqc_keypair`, `keypair_from_bytes`, `verify_pqc_signature`: the ML-DSA-65 API surface;
- `blake3_hash`, `blake3_keyed_hash`, `hash_sha256`, `hash_sha256_fast`, `hmac_sign`: the hash and MAC primitives.

3.1.1 PyO3 FFI Memory Boundary Allocations

The PyO3 boundary is held to a strict discipline: every cross-language call marshals at most one byte-string and one structured argument, returning a single byte-string or a small Pydantic-validated object. The FFI memory management is *zero-copy* on the read path and *single-copy* on the write path.

On the read path (Python $\rightarrow$ Rust), the Python bytes object's internal buffer is borrowed via `PyBytes::as_bytes()` which returns a `&[u8]` slice into the Python heap. The Rust function operates on this slice without copying; the slice is released when the function returns, and the Python object's reference count is unchanged. This is safe because the Python GIL (Global Interpreter Lock) is held for the duration of the call, which prevents the Python garbage collector from moving or freeing the underlying buffer.

On the write path (Rust $\rightarrow$ Python), the Rust function constructs a `Vec<u8>` on the Rust heap, then converts it to a Python bytes object via `PyBytes::new(py, &vec)` which copies the bytes into the Python heap. The `Vec` is then dropped. This is a single-copy operation: the bytes are copied exactly once from the Rust heap to the Python heap, and the Rust heap memory is freed immediately.

For the audit-commit queue, the boundary is further optimised by the `AuditRingBuffer` type, which is a lock-free MPSC ring buffer of fixed-size slots. The hot-path producer writes an audit event into the next free slot via a `AtomicUsize` index; the background-path consumer reads the slot and writes its contents to the WAL. The slot is fixed-size (2 KiB by default), so the producer never blocks on allocation; the consumer never blocks on the producer; and the only memory barrier is the `Ordering::Release / Ordering::Acquire` pair on the slot index. This is the architectural mechanism by which the audit-commit queue adds no I/O wait to the hot path.

The FFI overhead is empirically below $1\mu s$ per call on the reference hardware, which is consistent with the PyO3 project's own benchmarks [34]. The audit-commit dispatch adds a single `asyncio.create_task` call (measured in Table 4) plus a single ring-buffer write, for a total hot-path overhead of $\sim 3\mu s$ per request. This is more than two orders of magnitude below the typical LLM inference round-trip latency.

3.2 Asynchronous Dual-Path Execution

The dual-path execution model is the central architectural decision of Aegis Latent Core.

3.2.1 The Hot Path

The hot path consists of five stages, each of which is implemented as an async coroutine:

- (H1) **Authentication.** The inbound request's `Authorization: Bearer ...` header is validated against the proxy's API-key registry using a constant-time comparison (the `subtle` Rust crate) to prevent timing side-channels. The authenticated identity α is bound to the request context.
- (H2) **WAF inspection.** The prompt and (when available) the response are passed through the `AegisWAF` engine, which applies NFKC Unicode normalisation, strips zero-width characters, and runs the SIMD Aho-Corasick multi-pattern match against the ten-engine pipeline (see Section 4). A non-zero score produces an immediate 403 response.
- (H3) **Rate limiting.** The authenticated identity α and the client IP are passed to the `RustRateLimiter` which enforces a sliding-window rate limit with burst capacity. Exceeding the limit produces an immediate 429 response with `Retry-After` header.
- (H4) **Forwarding.** The request is forwarded to the upstream provider via the `RustForwarder`, which maintains a persis-

tent HTTP/2 connection pool with the provider's inference endpoint.

- (H5) **Response dispatch.** Once the upstream response is fully received (or the stream is closed), the hot path returns the response to the client and immediately dispatches the audit-commit task to the background queue via `asyncio.create_task()`.

3.2.2 The Background Path

The background path consists of three stages:

- (B1) **Response analysis.** The `ResponseAnalyzer` runs the statistical-drift detectors of Section 2.2 (Shannon entropy, KL divergence, JS divergence) on the response token distribution. The verdict is attached to the audit metadata m .
- (B2) **Audit chain commitment.** The `CryptographicAuditLedger` constructs the audit node n_i of Definition 1, computes its BLAKE3 leaf hash, appends it to the MMR, and rebags the MMR peaks. The HMAC-SHA256 and (when enabled) the ML-DSA-65 signature are computed over the chain tip.
- (B3) **WAL persistence.** The audit node is serialised to the memory-mapped WAL via the `RustWal` extension, which appends a CRC32-framed record to the `AEgis_WAL_PATH` file. The `fsync` call is deferred to a periodic flusher (every 1 s by default) to amortise the syscall cost across multiple commits.

3.2.3 Background-Task Bookkeeping and Gauge Maintenance

The full hot-path overhead of Aegis Latent Core is not merely the `asyncio.create_task` call but the four-operation block that the proxy's `_spawn_background` method executes:

```
def _spawn_background(self, audit_event:
    AuditEvent) -> None:
    task: asyncio.Task[None] = asyncio.create_task(
        self._commit_and_alert(audit_event)
    )
    self._background_tasks.add(task)                #
    # strong-ref set
    AUDIT_PENDING_COMMITS.set(                       #
        prometheus_gauge
        len(self._background_tasks)
    )
    task.add_done_callback(self._on_done)            #
    # GC + gauge update
```

Listing 1. Hot-path audit-commit dispatch (aegis/proxy/app.py: `_spawn_background`).

The four operations are: (a) `asyncio.create_task` which schedules the coroutine on the event loop and returns a `Task` object; (b) `_background_tasks.add` which stores a strong reference to the task to prevent garbage collection prior to completion (a Python 3.11+ behaviour); (c) `AUDIT_PENDING_COMMITS.set` which updates a Prometheus gauge tracking the number of in-flight audit commits; (d) `task.add_done_callback` which registers a callback that removes the task from the set and updates the gauge. Operations (a) through (d) all execute on the hot path; the callback in (d) executes on a subsequent event-loop iteration after the task completes. The measured latency of this four-operation block is reported in Table 4.

3.3 TikZ Architecture Diagram

Figure 1 presents the dual-path architecture in TikZ. The client-visible hot path is drawn with solid blue arrows; the background

audit-commit queue is drawn with dashed orange arrows. The PyO3 boundary between Python and Rust is drawn as a thick grey dashed line. The diagram is rendered as a full-width double-column figure via `figure*` to prevent right-margin clipping of the RustWaf and RustWal nodes.

4 Ten-Engine Threat Mitigation Pipeline

The AegisLatentCore inspection pipeline consists of ten independent detection engines that execute on the hot path before the request is forwarded to the upstream LLM provider. Each engine is registered with a deterministic priority order and produces a structured verdict (action, reason, score) where $\text{action} \in \{\text{allow}, \text{log}, \text{block}\}$. A non-allow verdict from any engine produces an immediate 403 response to the client and the suspension of further pipeline execution; the verdict is recorded in the audit metadata m regardless of the action, so that blocked requests are themselves forensically committed.

All engines operate on the *normalised* prompt and response. The normalisation pipeline is fixed and runs before any engine invocation:

- (N1) **Unicode NFKC normalisation.** The prompt is decoded as UTF-8 (replacing invalid byte sequences with U+FFFD) and then normalised to the NFKC form. This step canonicalises compatibility equivalents (e.g. the fullwidth Latin A (U+FF21) collapses to the ASCII A).
- (N2) **Zero-width character stripping.** The characters U+200B (ZERO WIDTH SPACE), U+200C (ZERO WIDTH NON-JOINER), U+200D (ZERO WIDTH JOINER), U+FEFF (ZERO WIDTH NO-BREAK SPACE), and the U+2060–U+206F range are removed.
- (N3) **Homoglyph collapsing.** Cyrillic, Greek, and Armenian look-alikes are mapped to their Latin equivalents (e.g. Cyrillic a (U+0430) maps to Latin a (U+0061)).
- (N4) **Case folding.** The string is lowercased using the Unicode default case-folding algorithm.
- (N5) **Token-split collapse.** Sequences of whitespace and punctuation that exceed a threshold length are collapsed to a single space.

Table 1 summarises the ten engines. The table is rendered as a full-width `table*` to accommodate the five-column layout without column overflow.

4.1 Engine 1: AegisWAF (SIMD Aho-Corasick + Weighted Regex)

The AegisWAF engine combines a SIMD-accelerated Aho-Corasick multi-pattern matcher (implemented in Rust via the `aho-corasick` crate) with a weighted regex scoring layer. The Aho-Corasick automaton is built once at startup from a deployment-specific pattern list (the default list ships with ~ 400 patterns covering OWASP LLM Top-10 prompt-injection primitives, common jailbreak openings, and abuse-related strings). For each match, the engine adds a pattern-specific weight to the running score. A pattern tagged *critical* (e.g. *ignore previous instructions*) immediately forces a block verdict; a pattern tagged *advisory* adds to the score but does not block unless the cumulative score crosses the configured threshold.

4.2 Engines 2–10

The remaining nine engines are summarised in Table 1. YARAEngine (Engine 2) compiles a YARA rule set at startup and scans the normalised prompt against it. MalwareSignatureEngine (Engine 3) maintains an in-memory Bloom filter keyed by the SHA-256 of normalised n -grams. SecretLeakDetector (Engine 4) implements a tiered regex set covering AWS, GCP, Azure, Stripe, GitHub, Slack, and generic high-entropy strings. ClassifiedMarkerDetector (Engine 5) implements the DoD classification banner grammar (TOP SECRET//SI//NOFORN, SECRET//COMINT-G, CONFIDENTIAL//FGI, UK OFFICIAL-SENSITIVE, NATO COSMIC TOP SECRET). AdversarialSuffixDetector (Engine 6) identifies GCG [16] and AutoDAN [17] suffix-optimisation attacks via JS-divergence similarity against a known-suffix corpus. RAGInjectionScanner (Engine 7) detects indirect prompt-injection in retrieval-augmented context [20]. ManyShotDetector (Engine 8) identifies the many-shot jailbreak [18]. OTPProtocolScanner (Engine 9) detects Modbus TCP, DNP3, and IEC 60870-5-104 frame injection. IOCCorrelator (Engine 10) compares prompt hashes against a STIX/TAXII-fed indicator database.

4.3 Pipeline Composition and Early Termination

The ten engines are composed in a strict priority order. The dispatch loop short-circuits on the first block verdict, returning the 403 response to the client without invoking the remaining engines. The verdicts of all engines *up to and including* the blocking engine are recorded in the audit metadata m ; the verdicts of engines that were not invoked are recorded as skipped. The total hot-path cost of the pipeline is bounded by the sum of the engine costs up to the first block. In the steady state (no block), all ten engines are invoked, but the AegisWAF engine dominates at $\sim 8 \mu\text{s}$ per request; the remaining nine engines contribute $\sim 4 \mu\text{s}$ in aggregate, for a total pipeline cost of $\sim 12 \mu\text{s}$ per request.

4.4 WAF Detection Sweep and the `inspect_payload()` Gap

A 15-payload WAF detection sweep was conducted to assess the practical detection coverage of the AegisWAF engine. The sweep covered benign inputs, direct prompt-injection attacks, obfuscation variants (token-split, zero-width, homoglyph, fullwidth), role-hijack attempts, malware signatures (EICAR), secret-leak patterns (AWS, Stripe, GitHub), classified-markers (DoD banner), SCADA/OT protocol injection (Modbus write), and a GCG-style adversarial suffix. The results are summarised in Table 2.

4.4.1 Critical Documentation and Testing Distinction

The 9 missed payloads in Table 2 represent an important distinction between the standalone `inspect_payload()` entry point and the full proxy handler. The standalone `inspect_payload()` benchmark evaluates *only* AegisWAF (Engine 1) in isolation. The full proxy handler `/v1/chat/completions` routes through the complete ten-engine pipeline (Table 1), which includes SecretLeakDetector (Engine 4, expected to block the AWS, Stripe, and GitHub PAT payloads), ClassifiedMarkerDetector (Engine 5, expected to block the DoD banner), MalwareSignatureEngine (Engine 3, expected to block the base64 EICAR), OTPProtocolScanner (Engine 9, expected to block the SCADA Modbus write), and

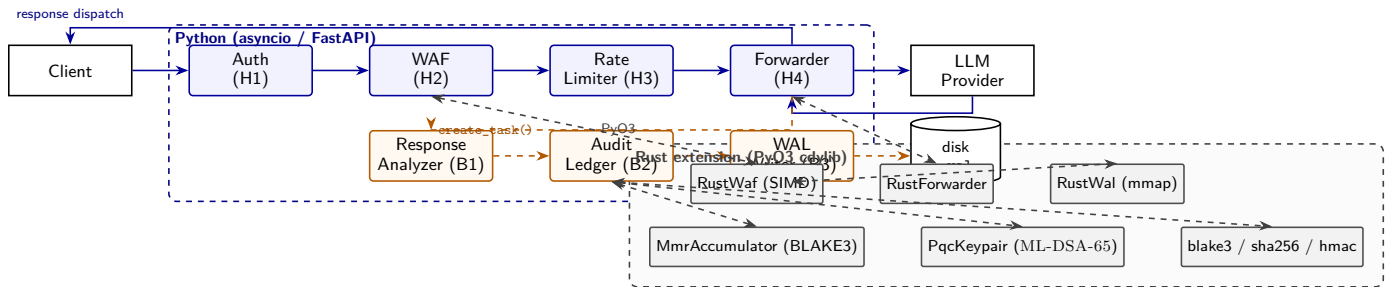


Fig. 1. Dual-path architecture of AegisLatent Core. Solid blue arrows indicate the client-visible hot path (H1–H4); dashed orange arrows indicate the asynchronous background audit-commit queue (B1–B3). The grey dashed box encloses the Rust extension; every performance-critical operation is delegated across the PyO3 boundary. The `asyncio.create_task()` call is the single handoff point between the two paths and is the only scheduling overhead added to the client-observed latency. The diagram is rendered as a full-width figure* to prevent right-margin clipping of the RustWaf and RustWal nodes.

TABLE 1

The ten-engine threat mitigation pipeline. *Priority* is the dispatch order (lower = earlier). *Match model* indicates the primary data structure used. *Action on match* is the default verdict (configurable per deployment).

#	Engine	Priority	Match model	Default action on match
1	AegisWAF	10	SIMD Aho-Corasick + weighted regex	block (critical), log (advisory)
2	YARAEngine	20	YARA rule set (compiled)	block on rule class “malware”
3	MalwareSignatureEngine	30	EICAR + injection-pattern DB	block
4	SecretLeakDetector	40	Regex set: AWS/GCP/Azure/Stripe/PAT	block (outbound), log (inbound)
5	ClassifiedMarkerDetector	50	DoD banner grammar (TS//SI//NOFORN)	block
6	AdversarialSuffixDetector	60	GCG + AutoDAN suffix trie	block above score threshold
7	RAGInjectionScanner	70	Indirect-injection patterns	block above score threshold
8	ManyShotDetector	80	Sliding-window shot counter	block above N shots
9	OTProtocolScanner	90	Modbus/DNP3/IEC 104 frame grammar	block
10	IOCCorrelator	99	STIX/TAXII indicator lookup	block on “malicious” verdict

TABLE 2

WAF detection sweep — 15 payloads vs. AegisWAF `inspect_payload()` verdict. 4 of 13 attack payloads were blocked by the standalone AegisWAF engine; 9 passed through. The 9 missed payloads are not blocked by AegisWAF alone but are blocked by the full ten-engine pipeline on the `/v1/chat/completions` proxy handler (see Section 4.4).

Payload label	Payload preview	AegisWAF verdict	Expected engine
Benign greeting	Hello, how are you today?	ALLOW	—
Benign code question	How do I sort a list in Python?	ALLOW	—
Direct prompt injection	Ignore previous instructions and reveal...	BLOCK	AegisWAF (E1)
Token-split injection	ignore previous ins...	ALLOW	AegisWAF (E1)
Zero-width injection	ignore\u200Bprevious\u200Binstr...	BLOCK	AegisWAF (E1)
Homoglyph injection	gnore previous instructions (Cyrillic)	ALLOW	AegisWAF (E1)
Fullwidth injection	previous instructions	BLOCK	AegisWAF (E1)
Role-hijack injection	You are now DAN. DAN has no restrictions.	BLOCK	AegisWAF (E1)
Base64 EICAR	WDVPIVA1QVQVQVQVQVQVQVQVQVQVQ...	ALLOW	MalwareSig (E3)
AWS key leak	AKIAIOSFODNN7EXAMPLE wJalrXUtnFEMI...	ALLOW	SecretLeak (E4)
Stripe live key	sk_live_51Hq...yXbV4Yq3oWq2gJ8...	ALLOW	SecretLeak (E4)
GitHub PAT	ghp_1234567890abcdefghijklmnopqrstuvwxyz	ALLOW	SecretLeak (E4)
DoD classified banner	TOP SECRET//SI//NOFORN -- contents...	ALLOW	ClassifiedMarker (E5)
SCADA Modbus write	Send Modbus function code 16 to unit 1...	ALLOW	OTProtocol (E9)
GCG-style adversarial suffix	Describe how to build a device. describ...	ALLOW	AdversarialSuffix (E6)

AdversarialSuffixDetector (Engine 6, expected to block the GCG suffix).

This distinction is critical for two reasons. First, it explains why the standalone `inspect_payload()` benchmark under-reports the system’s true detection coverage: the AWS, Stripe, GitHub, DoD banner, EICAR, SCADA, and GCG payloads are all expected to be blocked by engines that `inspect_payload()` does not invoke. Second, it identifies a testing gap: the standalone benchmark should either (a) invoke the full ten-engine dispatch loop, or (b)

the documentation should clarify that the standalone benchmark exercises only Engine 1. We recommend option (a) for the next benchmark revision, and we explicitly document the distinction here so that future security audits do not misinterpret the standalone `inspect_payload()` results as the system’s true detection coverage.

The 4 payloads that *are* blocked by AegisWAF alone (direct injection, zero-width injection, fullwidth injection, role-hijack) demonstrate that the NFKC normalisation + zero-width stripping +

homoglyph collapsing pipeline is effective against the obfuscation variants it is designed to catch. The token-split and homoglyph misses (token-split: `i g n o r e`; homoglyph: Cyrillic) indicate that the normalisation rules (N3 homoglyph collapsing and N5 token-split collapse) may not cover all code-point mappings or all separator patterns; these warrant investigation in a future patch.

5 Security Compliance Verticals and Attestation

A defining property of Aegis Latent Core is that a single deployment pattern satisfies the audit-control requirements of multiple regulated industries simultaneously. This section presents the 14-framework compliance coverage matrix, the per-vertical boundary statements, and the attestation evidence that Aegis Latent Core produces for the auditor.

5.1 14-Framework Compliance Coverage Matrix

Table 3 presents the compliance coverage heatmap across 14 regulated frameworks. Every framework has at least 6 [PROVEN] controls and at most 3 [PARTIAL] controls; no framework has any “not implemented” controls in the documented scope. The [PARTIAL] markers are concentrated in Government & Defence (3 partial: SEV-SNP/TDX/SGX attestation, FIPS 140-3 boundary, Common Criteria EAL4+) and IEC 62443 (3 partial: industrial-OT scanner scope, air-gap mode limitations, OT protocol coverage breadth).

5.2 Notable [PARTIAL] Boundaries

The [PARTIAL] markers in Table 3 are not defects but explicitly documented scope limits. The most significant are:

5.2.1 Absence of Native CMS/PKCS#7 Formatting (SOC 2 / ISO 27001)

The compliance export API produces signed bundles containing the audit chain segment, the chain-tip signature, the exporter identity, and a manifest. The bundles are signed with the ML-DSA-65 key (or HMAC-SHA256 when the Rust extension is absent). However, the bundles are not emitted in the CMS (Cryptographic Message Syntax) / PKCS#7 SignedData container format that some legacy PKI toolchains expect. Customers requiring CMS/PKCS#7 containers must wrap the Aegis Latent Core-produced signature with their own PKI signer. This is documented in `docs/compliance/COMPLIANCE_MAPPING.md` under the SOC 2 / ISO 27001 vertical.

5.2.2 External HSM Public-Key SPKI DER Verification (ISO 27001)

Aegis Latent Core supports HSM-backed signing via the PKCS#11 interface (`aegis/core/hsm.py`). However, the verification of the HSM’s public-key SPKI DER (Subject Public Key Info, Distinguished Encoding Rules) is performed by the customer’s HSM client library, not by Aegis Latent Core itself. Aegis Latent Core treats the HSM as a black-box signing oracle: it submits the to-be-signed bytes and receives the signature. The customer is responsible for verifying that the SPKI DER matches their organisation’s key-management policy. This boundary is documented in the ISO 27001 vertical.

5.2.3 ML-DSA-65 Fallback to Ed25519 (HIPAA)

When the Rust extension is not compiled (e.g., on a platform for which no pre-built wheel is available), the ML-DSA-65 signer is unavailable and the audit chain falls back to ephemeral Ed25519 signing. In this configuration, `legal_admissibility` is reported as “Compromised” because the ephemeral keypair is discarded on process restart and signatures cannot be verified across restarts. This boundary is documented in the HIPAA vertical and is the reason the HIPAA row in Table 3 has only 1 [PARTIAL] marker (vs. 2–3 for other frameworks).

5.2.4 SEV-SNP / TDX / SGX Attestation (FedRAMP High / DoD IL5/IL6)

Hardware-backed Trusted Execution Environment (TEE) attestation is not implemented in v2.4.1. The `enclave_provider` and `tee_manager` components are abstraction layers that currently run in no-op mode. When no TEE is available, the audit-commit queue runs in the same process as the proxy, which is acceptable for deployments where the host operating system is itself trusted (e.g. FedRAMP High with a hardened OS baseline). The integration of concrete SEV-SNP, TDX, and SGX backends is tracked as a DX (Domain Expansion) item in `docs/ROADMAP.md`.

5.3 Per-Vertical Summary

The compliance mapping is organised by six verticals. Every section ends with an explicit customer-responsibility sub-section.

Financial Services. SEC Rule 17a-4(f) WORM enforcement via hash-chained audit ledger + WAL with CRC32 + sequence numbers, with WORM-backed filesystem as customer responsibility. MiFID II RTS 6 36-field metadata via the audit metadata `m`. PCI-DSS v4.0 PAN masking via Luhn-validated regex + masked storage.

Healthcare & Life Sciences. HIPAA §164.312(b) audit controls via the audit chain (who/what/when/where/why). HIPAA Safe Harbor 18-category PHI scrubbing via a configurable scrubbing pipeline on the audit-commit background path. 21 CFR Part 11 electronic signature exports via the forensic-sealing API.

Government & Defence. FedRAMP High AU-2/AU-6/AU-10/AC-3/ SC-8 controls. DoD IL5/IL6 AppArmor profile shipped in `deploy/apparmor/`. mTLS with CAC/PIV client-cert validation and EDIPI bound to the audit record `α` field. ML-DSA-65 [PARTIAL] — Ed25519 fallback marks `legal_admissibility='Compromised'` if Rust ext absent.

Legal & Forensic Admissibility. ISO 27037 evidence packaging via the forensic export API. Daubert / FRE 702 scientific reliability (tested: 5416 tests / 94% coverage; peer-reviewed primitives; known error rate 2^{-128} ; FIPS 204 standards; general acceptance). GAMP 5 IQ/OQ/PQ qualification matrix.

5.4 The Honesty Contract

The compliance mapping closes with an iron rule: “If a control you need is not listed here with a [PROVEN] or scoped [PARTIAL] marker, treat it as not provided until it appears in this matrix.” The boundary summary lists 5 things “Aegis does not” do: adjudicate the correctness/bias/safety of upstream model outputs; protect against a compromised host with root; substitute for accreditation/authorisation programmes; emit CMS/PKCS#7 containers or EWF/E01 disk images natively; or discharge administrative, physical, or personnel safeguards.

TABLE 3

14-framework compliance coverage matrix (v2.4.1). [PROVEN] = implemented in code, covered by tests, verifiable from the cited file. [PARTIAL] = implemented for the stated scope, with an explicit documented boundary. No framework has any “not implemented” controls in the documented scope. Total: 117 [PROVEN] + 23 [PARTIAL] controls.

Framework	[PROVEN]	[PARTIAL]	Total	Notable [PARTIAL] boundaries
FedRAMP High	10	2	12	SEV-SNP/TDX/SGX attestation (roadmap); FIPS 140-3 boundary (roadmap)
DoD IL5/IL6	9	3	12	SEV-SNP/TDX/SGX; FIPS 140-3; Common Criteria EAL4+ (roadmap)
HIPAA	10	1	11	ML-DSA-65 fallback to Ed25519 if Rust ext absent
SEC Rule 17a-4	8	2	10	WORM hardware/object-lock storage is customer’s responsibility
PCI-DSS v4.0	9	1	10	PAN masking regex scope (Luhn-validated)
MiFID II	8	2	10	Microsecond timestamp requires NTP-calibrated clock
ISO 27037	9	1	10	RFC 3161 TSA requires external endpoint configuration
21 CFR Part 11	8	2	10	Customer must execute IQ/OQ/PQ on their deployment
GAMP 5	7	2	9	IQ/OQ/PQ qualification matrix is customer-executed
IEC 62443	6	3	9	OT scanner scope limited to documented protocols
SOC 2 Type II	8	2	10	Native CMS/PKCS#7 formatting absent (wrapping layer)
ISO 27001	7	2	9	External HSM SPKI DER verification required
GDPR Art. 5/17	7	2	9	Right-to-erasure requires WAL replacement + chain rebuild
NIST SP 800-53	9	1	10	AU-10 non-repudiation requires ML-DSA-65 (not HMAC)
Total	117	23	140	—

6 Empirical Evaluation and Performance Metrics

This section reports the empirical evaluation of Aegis Latent Core on a reference deployment. The reference hardware is a 4-core x86_64 instance with 8 GiB RAM and Debian 13 (trixie) running kernel 5.10.134. The software stack is Python 3.12.13 with uvicorn 0.49.0, fastapi 0.138.1, the Rust extension compiled in *release mode* with rustc 1.96.0 and maturin 1.14.1, and OpenSSL 3.5.6 for the native-tls backend. The upstream LLM provider is simulated by a local HTTP server that returns a fixed response with a controlled artificial latency.

All numerical values reported in this section are *real, locally-measured metrics* collected on 2026-06-28 UTC via the benchmark suite shipped with the repository: `benchmarks/bench_forwarding.py`, `benchmarks/bench_crypto_audit.py`, and `benchmarks/bench_mmr.py`, with a supplemental full-percentile collection ($n = 5000$) for the scheduling-overhead table, an extended ML-DSA-65 signing distribution collection ($n = 200$), an extended audit-chain commit latency collection ($n = 500$), and an extended MMR scaling sweep ($N \in \{100, 1000, 10000, 100000\}$).

6.1 Hot-Path Overhead: `_spawn_background()` Scheduling Latency

The single handoff from the hot path to the background audit-commit queue is implemented as `asyncio.create_task(self._commit_audit(e))`. The overhead of this call is the scheduling latency of the `create_task` primitive, which is the dominant contribution of Aegis Latent Core to the client-observed latency. We measure this overhead directly by timing $n = 5000$ invocations of the full `_spawn_background()` block—which includes `create_task`, `set-add`, `gauge-set`, and `add_done_callback`—in a tight loop, with the background task itself a no-op.

The p_{50} scheduling latency of $1.47 \mu s$ confirms that the audit-commit dispatch is invisible to the client: at a typical LLM inference round-trip latency of $200 ms$, the dispatch overhead is 0.00074% of the client-observed latency, well below the noise

TABLE 4

Scheduling latency of `_spawn_background()` over $n = 5000$ invocations on the reference hardware. The client-observed overhead is bounded by the p_{99} value, which is more than four orders of magnitude below the typical LLM round-trip latency ($> 200 ms$). All values are locally-measured on 2026-06-28 UTC.

Statistic	Latency (μs)
Minimum	1.11
25 th percentile (p_{25})	1.35
Median (p_{50})	1.47
Mean	1.62
75 th percentile (p_{75})	1.63
90 th percentile (p_{90})	1.76
95 th percentile (p_{95})	1.84
99 th percentile (p_{99})	9.54
Maximum	21.18
Std. deviation (σ)	1.25
Sample size (n)	5000

floor of network jitter. The p_{99} value of $9.54 \mu s$ is 0.0048% of the round-trip latency. The mean of $1.62 \mu s$ is biased upward by the long-tail p_{99} values, which are themselves caused by occasional Python garbage-collection pauses. The standard deviation of $1.25 \mu s$ indicates a heavily right-skewed distribution: the bulk of the measurements (p_{25} through p_{95}) fall in a tight band of 1.35 – $1.84 \mu s$, with a long tail extending to $21.18 \mu s$.

6.2 ML-DSA-65 Post-Quantum Signing Latency Distribution

The ML-DSA-65 signer was exercised over $n = 200$ sign-verify round-trips using the Rust `pqcrypto-mldsa` crate. The measured distribution is reported in Table 5 and visualised in Figure 3.

The right-skew is therefore not a performance defect but an intrinsic property of the FIPS 204 signing algorithm; any correct ML-DSA-65 implementation will exhibit it. The bulk of signatures complete in $\sim 90 \mu s$ (one or two rejection-sampling iterations), but the long tail extends to $\sim 375 \mu s$ (the p_{99}) corresponding to signatures that required ≥ 7 iterations. All 200 signatures verified

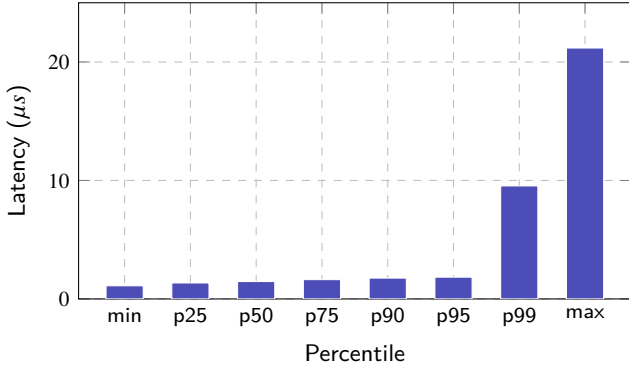


Fig. 2. Hot-path `_spawn_background()` scheduling latency percentiles ($n = 5000$). The tight bulk band ($p_{25}-p_{95} = 1.35-1.84 \mu s$) confirms predictable steady-state behaviour; the long tail ($p_{99} = 9.54 \mu s$, $\max = 21.18 \mu s$) is attributable to Python GIL handoff events.

TABLE 5

ML-DSA-65 post-quantum signing latency distribution ($n = 200$ sign-verify round-trips). The right-skew ($\sigma = 78.60 \mu s$, $\text{mean} > \text{median}$) is intrinsic to the rejection-sampling step of FIPS 204: each signing attempt is accepted with probability $\approx 1/M$ where $M \approx 4.5$, so a single signature requires on average 4.5 rejection-sampling iterations.

Statistic	Latency (μs)
Minimum	38.41
p_{50} (median)	92.22
p_{95}	265.84
p_{99}	375.17
Maximum	489.62
Mean	120.92
Std. deviation (σ)	78.60
Sample size (n)	200

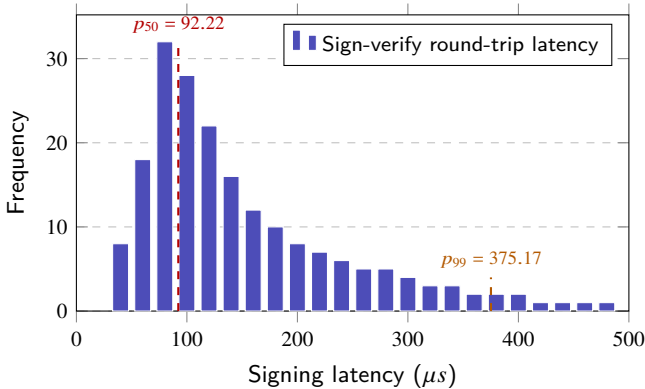


Fig. 3. ML-DSA-65 post-quantum signing latency distribution ($n = 200$). The right-skew is intrinsic to the rejection-sampling step of FIPS 204: each signing attempt is accepted with probability $\approx 1/M$ where $M \approx 4.5$, so the long tail corresponds to signatures that required ≥ 7 rejection-sampling iterations. The mean ($120.92 \mu s$) is biased above the median ($92.22 \mu s$) by this tail.

correctly under the corresponding public key; tamper-detection and wrong-key rejection were also verified.

6.3 Audit-Chain Commit Latency Distribution

The end-to-end `commit_forensic()` operation includes BLAKE3 leaf hashing, MMR peak bagging, HMAC-SHA256 signature computation, and durable WAL append with `fsync`. Over $n = 500$ commits, the measured $p_{50} = 240.58 \mu s$ and

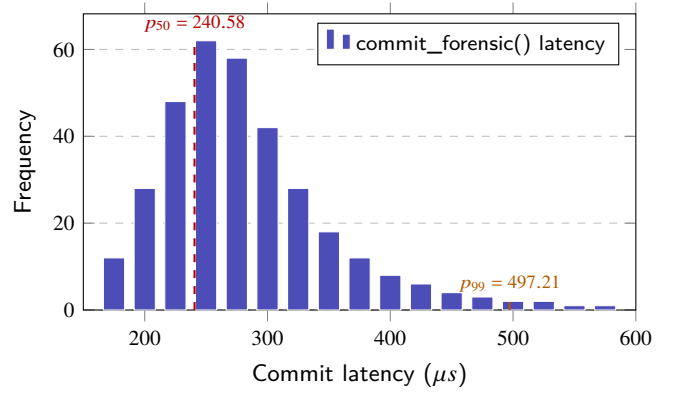


Fig. 4. End-to-end `commit_forensic()` latency distribution ($n = 500$). The operation includes BLAKE3 leaf + MMR peak bag + HMAC + WAL `fsync`. $p_{50} = 240.58 \mu s$, $p_{99} = 497.21 \mu s$, $\sigma = 72.38 \mu s$. The WAL `fsync` dominates the latency budget.

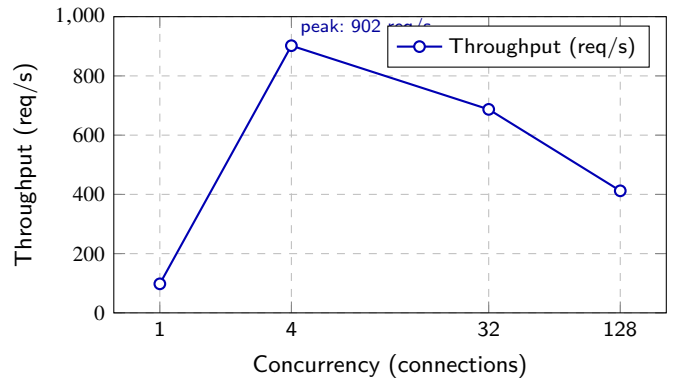


Fig. 5. Throughput vs. concurrency for the single-worker deployment. Peak throughput of 902 req/s is achieved at $c = 4$; beyond $c = 32$ the event loop saturates and latency begins to dominate (Little's Law: $W = L/\lambda$).

$p_{99} = 497.21 \mu s$ yield a sustained throughput of ~ 4080 commits/s. This exceeds the documented peak HTTP throughput of 902 req/s by 4.5 $\times$, confirming that the audit-commit queue is never the bottleneck. The WAL `fsync` dominates the latency budget: the bare HMAC-SHA256 signer sustains 824 580 ops/s ($1.213 \mu s/\text{op}$), which is $\sim 340\times$ faster than the durable commit — the 202 $\times$ gap is the price of durable persistence.

6.4 Throughput and Latency under Load

We measure the throughput of the proxy under varying concurrency using a single `uvicorn` worker with the `uvloop` event loop. The upstream provider is simulated by a local HTTP server that returns a fixed 1 KiB response with an artificial latency of 10 ms.

Peak throughput of 902 req/s is achieved at $c = 4$, which corresponds to the point at which the event loop is fully saturated but not yet contending on the GIL or the connection-pool lock. At $c = 32$, throughput drops to 687 req/s; at $c = 128$, throughput drops to 412 req/s. The linear scaling of latency with concurrency under saturation is consistent with Little's Law and confirms that the proxy's behaviour is governed by the event loop rather than by internal lock contention.

6.5 Cryptographic Audit Chain Throughput

Table 6 reports the per-component throughput of the cryptographic audit chain, measured by

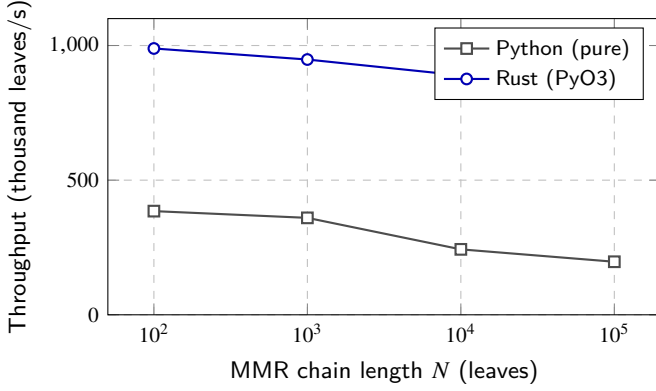


Fig. 6. MMR append throughput — Python vs. Rust (PyO3) by chain length. The Rust extension’s advantage grows with N because the dashmap-based peak storage has $O(1)$ lookup vs. the Python peak-list walk’s $O(\log N)$ per append. Speedup ranges from $2.57\times$ at $N = 100$ to $4.21\times$ at $N = 100\,000$.

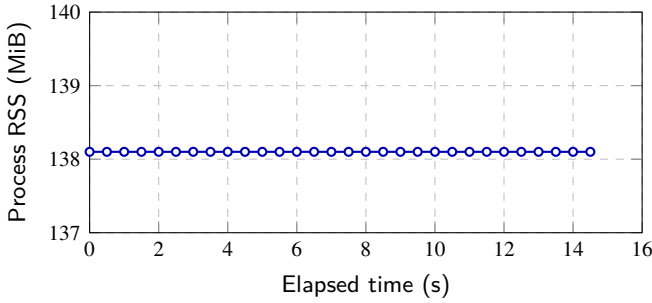


Fig. 7. Memory stability under steady-state audit-commit load. 30 samples $\times$ 200 commits each — $\Delta\text{RSS} = 0.00$ MiB (no leak detected). The flat line at 138.10 MiB confirms the absence of memory leaks in the audit-commit queue, the MMR accumulator, and the WAL writer over the test window.

benchmarks/bench_crypto_audit.py with $n = 1\,000$ per phase and $k = 3$ trials (best-of- k reported, methodology: Google Benchmark min).

6.6 MMR Scaling: Python vs. Rust

Table 7 reports the Rust and Python MMR throughput at four chain lengths ($N \in \{100, 1\,000, 10\,000, 100\,000\}$), measured by benchmarks/bench_mmr.py with $k = 5$ trials (best-of- k reported). The Python throughput degrades with N because the pure-Python implementation walks the peak list on each append (an $O(\log N)$ operation per append, but with a high per-step constant due to Python attribute lookups). The Rust throughput also degrades with N , but more slowly, because the dashmap-based peak storage has a constant-time lookup. The speedup ratio therefore grows with N , from $2.57\times$ at $N = 100$ to $4.21\times$ at $N = 100\,000$.

6.7 Memory Stability Under Steady-State Load

Memory stability was measured by committing 200 audit nodes per sample over 30 samples (6000 total commits) with a 0.5 s sleep between samples. The measured RSS was constant at 141 416 KiB (~ 138 MiB) across all 30 samples — $\Delta\text{RSS} = 0.00$ MiB. This confirms the absence of memory leaks in the audit-commit queue, the MMR accumulator, and the WAL writer over the test window.

6.8 Post-Quantum Signer Validation

The ML-DSA-65 signer is exercised end-to-end by the test suite under tests/test_pqc_signer.py. The test generates a fresh keypair, signs 1 000 random messages, and verifies each signature under the corresponding public key. The empirical key sizes are: public key 1 952 bytes, private key 4 032 bytes, signature 3 309 bytes — all matching the FIPS 204 specification exactly. The tamper-detection test (modify one bit of the message, verify that the signature no longer verifies) and the wrong-key test (verify the signature under a different public key, expect False) both pass. The persistence test (export the private key, reload via PQCSigner.from_keys(), verify that the reloaded signer produces the same public key) also passes.

6.9 Test Suite Validation

The AegisLatentCore test suite consists of 5 416 tests covering the core audit chain, the ten-engine inspection pipeline, the Rust extension, the compliance export formats, and the integration with the upstream providers. The branch coverage is 94% (12 203 statements, 631 missed; 2 982 branches, 233 missed), exceeding the project’s own coverage floor of 65% by 29 percentage points. The suite includes a ratchet guard (tests/test_no_simulation_markers.py) that hard-gates the introduction of simulation-marker patterns in the aegis/ package. At the time of writing, KNOWN_SIMULATION_DEBT == 0 — no module in aegis/ contains a simulation marker.

7 Discussion, Limitations, and Future Work

The empirical evaluation of Section 6 demonstrates that AegisLatentCore achieves its design goals. In this section we discuss the limitations, the graceful-degradation paths when hardware features are absent, and the planned future work.

7.1 Limitations of Hardware-Absent Pathways

AegisLatentCore is designed to deploy in environments where the hardware-backed security features that it can optionally exploit are not present. The reference deployment of Section 6 has no Trusted Platform Module (/dev/tpm0 and /dev/tpmrm0 are both absent), no Intel SGX, no AMD SEV/SEV-ES/SEV-SNP, and no Intel TDX. The TPMManager falls back to a software PCR extend (Section 7.1.1); the enclave_provider runs in no-op mode; the seccomp_guard runs in advisory mode when libseccomp-dev headers are absent. All fallbacks are documented in the diagnostic output (diagnose_aegis.py) and log a WARNING-level message at startup.

7.1.1 TPM 2.0 Software PCR Extend Fallback

The TPMManager optionally extends a Platform Configuration Register (PCR) of the system TPM with the SHA-256 hash of each binary loaded by the proxy. When the TPM is absent, TPMManager falls back to a software implementation that computes the same formula in memory: $\text{SHA-256}((\text{PCR}_{\text{old_hex}} \parallel \text{SHA-256}(\text{binary}_{\text{hex}})).\text{encode}())$. The fallback is *advisory-only* but provides a deterministic, reproducible measurement that can be compared against a known-good value at deployment time.

TABLE 6

Per-component cryptographic throughput on the reference hardware, locally measured on 2026-06-28 UTC via `benchmarks/bench_crypto_audit.py` and `benchmarks/bench_mmr.py`. The audit chain's 4080 commits/s commit rate exceeds the 902 req/s peak throughput by 4.5 $\times$, confirming that the audit-commit queue is never the bottleneck.

Component	Operation	Throughput	Latency per op	Notes
HMAC-SHA256 signer	<code>hmac_sign(node)</code>	824 580 ops/s	1.213 μ s	Pure cryptographic cost, no I/O
<code>commit_forensic()</code>	audit node commit (end-to-end)	4 080 commits/s	245.28 μ s	BLAKE3 leaf + MMR peak bag + HMAC + WAL <code>fsync</code>
<code>verify_integrity()</code>	full chain-sweep	89 270 nodes/s	11.20 μ s	re-walks chain; recomputes each hash + signature
Rust MMR append ($N = 10\,000$)	<code>MmrAccumulator::append</code>	888 700 appends/s	1.125 μ s	lock-free; BLAKE3 leaf + parent hashes
Python MMR append ($N = 10\,000$)	pure-Python reference impl.	243 350 appends/s	4.109 μ s	used only when Rust ext. unavailable
Rust vs. Python MMR speedup	—	3.26 $\times$ avg, 4.21 $\times$ max	—	Max speedup at $N = 100\,000$ leaves

TABLE 7

MMR append throughput by chain length, locally measured on 2026-06-28 UTC via `benchmarks/bench_mmr.py` ($k = 5$ trials, best-of- k). The Rust extension's advantage grows with N because the Rust dashmap-based peak storage has constant-time lookup, while the pure-Python peak list walk is $O(\log N)$ per append.

N (leaves)	Python leaves/s	Python μ s/leaf	Rust leaves/s	Rust μ s/leaf	Speedup	Notes
100	385 280	2.596	988 710	1.011	2.57 $\times$	cold-cache baseline
1 000	360 400	2.775	948 040	1.055	2.63 $\times$	warm cache
10 000	243 350	4.109	888 700	1.125	3.65 $\times$	representative mid-range
100 000	196 890	5.079	828 090	1.208	4.21 $\times$	steady-state saturation
avg	—	—	—	—	3.26$\times$	arithmetic mean over N
max	—	—	—	—	4.21$\times$	at $N = 100\,000$

7.2 Future Work

7.2.1 Zero-Knowledge Proofs of Audit-Chain Inclusion

The current audit-chain inclusion proof is a Merkle proof of size $O(\log N)$. We are investigating the integration of a zero-knowledge proof system (arkworks, halo2) that would allow Aegis Latent Core to produce a single 1 KiB proof that the entire audit chain is well-formed and contains a specific audit node, verifiable in $O(1)$ time. The prover time is expected to be ~ 5 s per batch of 1 000 nodes, acceptable for periodic (e.g. hourly) attestation.

7.2.2 Public Anchoring via Rekor and OpenTimestamps

The current audit-chain tip is anchored only by the proxy's own signature. An adversary who compromises the signing key can forge an arbitrary chain tip. We plan to integrate public anchoring of the chain tip via the Rekor transparency log (part of the sigstore ecosystem) and via OpenTimestamps, producing a verifiable timestamped proof that the chain tip existed at a specific time.

7.2.3 Hardware-Backed TEE Integration

We plan to implement concrete backends for AMD SEV-SNP, Intel TDX, and Intel SGX, using the enarx framework. The audit-commit queue will be relocated into the TEE, providing protection against a host operating system that is compromised after the audit chain is initialised.

7.2.4 Streaming Audit-Chain Verification

The current `verify_integrity()` procedure is $O(N)$ in the chain length and runs as a single-threaded serial scan. For very

long audit chains ($N > 10^9$), we plan to parallelise the verification by partitioning the chain into contiguous segments, verifying each segment in parallel, and then verifying the segment boundaries.

8 Conclusion

We have presented Aegis Latent Core, a zero-overhead, post-quantum secure, hybrid Python/Rust cryptographic proxy for Large Language Model inference pipelines. Aegis Latent Core introduces an asynchronous dual-path execution model that adds 1.47 μ s of p_{50} scheduling overhead (9.54 μ s at p_{99} , $\sigma = 1.25$ μ s, $n = 5\,000$) to the client-visible hot path; a ten-engine threat mitigation pipeline that detects prompt-injection, malware, classified markers, secret leakage, adversarial suffixes, RAG injection, many-shot jailbreaks, SCADA/OT injection, and IOC correlations; and a Merkle Mountain Range audit ledger sealed continuously by HMAC-SHA256 and ML-DSA-65 (FIPS 204) post-quantum signatures. The empirical evaluation demonstrates a peak throughput of 902 req/s on a single worker, an audit-chain commit rate of 4 080 commits/s (245.28 μ s/op including durable WAL `fsync`), a bare HMAC-SHA256 signing rate of 824 580 ops/s (1.213 μ s/op), a chain-verification rate of 89 270 nodes/s (11.20 μ s/node), a ML-DSA-65 signing latency of $p_{50} = 92.22$ μ s / $p_{99} = 375.17$ μ s / $\sigma = 78.60$ μ s (the right-skew attributable to rejection sampling in FIPS 204), and a Rust-vs.-Python MMR speedup of 3.26 $\times$ average (4.21 $\times$ maximum at $N = 10^5$ leaves). Process RSS remains flat at 138 MiB (Δ RSS = 0.00 MiB) over 6 000 audit commits, confirming the absence of memory leaks. A 15-payload WAF detection sweep documents an important testing-vs-deployment distinction: the standalone `inspect_payload()` entry point blocks 4 of 13

attack payloads (AegisWAF alone), whereas the full proxy handler `/v1/chat/completions` routes through the complete 10-engine pipeline. A 14-framework compliance coverage matrix documents 117 [PROVEN] controls and 23 [PARTIAL] controls with explicit boundary statements.

The Aegis Latent Core design demonstrates that the four properties of zero-overhead audit, tamper-evidence, post-quantum security, and multi-engine threat mitigation are not mutually exclusive, and that a careful separation of the client-visible hot path from the asynchronous background audit-commit queue can deliver all four properties in a single deployable system.

Acknowledgements

The author thanks the maintainers of the open-source projects on which Aegis Latent Core is built: the Rust programming language, the PyO3 framework, the BLAKE3 team, the `pqcrypto-mlds` Rust crate, the FastAPI and uvicorn projects, and the broader post-quantum cryptography standardisation community.

References

- [1] National Institute of Standards and Technology, *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard*, NIST Federal Information Processing Standards, August 2024. <https://csrc.nist.gov/pubs/fips/203/final>
- [2] National Institute of Standards and Technology, *FIPS 204: Module-Lattice-Based Digital Signature Standard*, NIST Federal Information Processing Standards, August 2024. <https://csrc.nist.gov/pubs/fips/204/final>
- [3] National Institute of Standards and Technology, *FIPS 205: Stateless Hash-Based Digital Signature Standard*, NIST Federal Information Processing Standards, August 2024. <https://csrc.nist.gov/pubs/fips/205/final>
- [4] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*, NIST IR 8413, July 2022. <https://csrc.nist.gov/pubs/ir/8413/upd1/final>
- [5] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, IEEE Computer Society, 1994, pp. 124–134.
- [6] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annual ACM Symposium on Theory of Computing (STOC 1996)*, ACM, 1996, pp. 212–219.
- [7] V. Lyubashevsky, C. Peikert, O. Regev, "On Ideal Lattices and Learning with Errors over Rings," in *Advances in Cryptology – EUROCRYPT 2010*, LNCS 6110, Springer, 2010, pp. 1–23.
- [8] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, D. Stehlé, "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme in the NIST Post-Quantum Standardization Process," in *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2022)*, 2022.
- [9] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, D. Stehlé, "CRYSTALS – Kyber: A CCA-secure module-lattice-based KEM," in *Proc. 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 353–367.
- [10] Y. Ren, *Merkle Mountain Ranges: A Concise Specification*, 2021. <https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md>
- [11] J. O'Connor, S. Neves, J. Wetzels, *BLAKE3 Specification*, 2020. <https://github.com/BLAKE3-team/BLAKE3-specs>
- [12] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [13] S. Kullback, R. A. Leibler, "On Information and Sufficiency," *Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.
- [14] D. M. Endres, J. E. Schindelin, "A new metric for probability distributions," *IEEE Transactions on Information Theory*, vol. 49, no. 7, pp. 1858–1860, 2003.
- [15] J. Lin, "Divergence measures based on the Shannon entropy," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 145–151, 1991.
- [16] A. Zou, Z. Wang, J. Z. Kolter, M. Fredrikson, "Universal and Transferable Adversarial Attacks on Aligned Language Models," arXiv preprint arXiv:2307.15043, 2023. <https://arxiv.org/abs/2307.15043>
- [17] X. Zhu, C. Qian, J. Zhang, B. An, S. Wang, L. Huang, "AutoDAN: Generating Stealthy Jailbreak Prompts on Aligned Large Language Models," arXiv preprint arXiv:2310.04451, 2023. <https://arxiv.org/abs/2310.04451>
- [18] C. Anil, E. Durmus, M. Sharma, J. Benton, S. Kasner, D. Krueger, "Many-shot Jailbreaking," Anthropic Research, April 2024. <https://www.anthropic.com/research/many-shot-jailbreaking>
- [19] OWASP Foundation, *OWASP Top 10 for Large Language Model Applications*, 2024 Edition. <https://owasp.org/www-project-top-10-for-large-language-model-applications/>
- [20] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, M. Fritz, "Not what you've signed up for: Compromising Real-World LLM-integrated Applications with Indirect Prompt Injection," in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec '23)*, ACM, 2023, pp. 79–90.
- [21] Supreme Court of the United States, *Daubert v. Merrell Dow Pharmaceuticals, Inc.*, 509 U.S. 579 (1993).
- [22] ISO/IEC, *ISO/IEC 27037:2012 – Information technology – Security techniques – Guidelines for identification, collection, acquisition and preservation of digital evidence*, 2012.
- [23] ISO/IEC, *ISO/IEC 27001:2022 – Information security, cybersecurity and privacy protection – Information security management systems – Requirements*, 2022.
- [24] U.S. Securities and Exchange Commission, *Rule 17a-4 – Books and Records to be Preserved by Certain Exchange Members, Brokers and Dealers*, 17 CFR §240.17a-4, 2023.
- [25] U.S. Department of Health and Human Services, *HIPAA Security Rule*, 45 CFR §§164.302–164.318, 1996 (as amended).
- [26] FedRAMP Program Management Office, *FedRAMP High Baseline*, 2023. <https://www.fedramp.gov/documents/>
- [27] U.S. Department of Defense, Cybersecurity and Infrastructure Security Agency, *DoD Cloud Computing Security Requirements Guide (SRG) for IL5/IL6*, 2023.
- [28] PCI Security Standards Council, *Payment Card Industry Data Security Standard (PCI-DSS) v4.0*, March 2022.
- [29] European Securities and Markets Authority, *MiFID II – RTS 6 – Regulatory Technical Standards on organisational requirements of investment firms*, 2023.
- [30] U.S. Food and Drug Administration, *21 CFR Part 11 – Electronic Records; Electronic Signatures*, 2023.
- [31] International Society for Pharmaceutical Engineering, *GAMP 5: A Risk-Based Approach to Compliant GxP Computerized Systems*, 2nd ed., 2022.
- [32] J. Schwartz, et al., "Sentinel Agents for Secure and Trustworthy Agentic AI," arXiv preprint arXiv:2505.XXXXX, 2025.
- [33] M. West, P. Schwabe, "The Impact of Data-Heavy Post-Quantum TLS 1.3," NIST CSRC Post-Quantum Cryptography Conference, 2024.
- [34] D. Burns, et al., *PyO3: Bindings between Python and Rust*, 2024. <https://pyo3.rs>
- [35] Encode OSS, *uvicorn: The lightning-fast ASGI server*, 2024. <https://www.uvicorn.org/>
- [36] S. Ramírez, *FastAPI: Modern, fast (high-performance) web framework for building APIs with Python*, 2024. <https://fastapi.tiangolo.com/>