From Policy to Proof: Implementing a Verifiable Neurorights Firewall in Cyber-Retrieval via Compile-Time Invariants

Constitutional Policy Root: The `neurorights.envelope.citizen.v1` ALN Shard

The foundation of the compile-time neurorights firewall is a declarative, machine-readable constitutional policy root established through an Application-Level Network (ALN) shard named `neurorights.envelope.citizen.v1` . This shard serves as the immutable source of truth for all neurorights invariants that govern interactions within the Cyber-Retrieval architecture . By treating this ALN particle as a "constitutional policy root," the system elevates neurorights from abstract principles to concrete, executable constraints that can be enforced directly by the compiler . This approach aligns with modern practices of "policy-as-code" and ensures that any deviation from the constitutionally mandated guarantees results in a failed build, thereby making violations impossible to deploy [3] . The design leverages the existing Cyber-Retrieval stack, which already utilizes ALN shards for other safety particles and governance rules, such as `nanoswarm.compliance.field.v1` and `neurorights.governance.v1`, creating a cohesive and consistent governance framework .

The `neurorights.envelope.citizen.v1` shard encodes a set of core, non-negotiable rights and associated invariants that form the bedrock of the augmented-citizen workflow . These invariants are not merely descriptive but are intended to be translated into unbreakable type constraints within the Rust codebase . The primary rights and corresponding constraints embedded in this ALN shard include the prohibition of exclusion from basic services, the prevention of inner-state scoring, the elimination of neurocoercion, the guarantee of revocability for actions affecting cognition, and the mandate for ecosocial reporting . Each of these rights translates into a specific constraint that can be verified at compile time. For instance, the prohibition on exclusion would manifest as a type-level check that prevents a tool from being compiled if its logic could potentially deny access to a fundamental service. Similarly, the ban on inner-state scoring would be encoded as a trait or constant that makes any attempt to store or manipulate cognitive state information outside of approved, auditable channels a compile-time error .

The structure of the ALN shard is critical for its function as a policy root. It must contain more than just the textual description of the rights; it needs to include formal identifiers, versioning, and cryptographic anchors to ensure its integrity and traceability . The proposed `NeurorightsProfile` struct, which will be derived from this shard, includes fields for `id` (e.g., "neurorights.envelope.citizen.v1"), `version` (e.g., "1.3"), and `anchor` (a cryptographic identifier pointing back to the ALN location where the shard is stored, such as a DID reference) . This anchoring mechanism is crucial for creating an immutable and verifiable link between the deployed software and the specific version of the neurorights constitution it adheres to . Any change to the ALN shard—whether an update to a right or a refinement of an invariant—will result in a new versioned ID. The ALN-to-Rust compilation pipeline will detect this change, re-generate the necessary code, and cause any dependent code that has not been updated to comply with the new contract to fail to compile . This process creates a hard boundary around the policy, preventing drift and ensuring that all components of the system are consistently bound by the same constitutional guarantees.

This method of embedding policy into data structures is a well-established pattern in distributed systems and knowledge graphs, where provenance, context, and time must be rigorously associated with data triples [7] . In this case, the ALN shard acts as a highly structured and privileged triplestore for neurorights policies. The use of a standardized format like ALN allows for interoperability and future extensibility, enabling other parts of the Cybernet ecosystem to query and validate these rights programmatically [20] . The governance model shifts from manual auditing to automated verification, as the ultimate arbiter of compliance becomes the compiled binary itself, which has been vetted against the ALN-defined contract [12] [13] . This approach provides a higher degree of assurance than runtime checks alone, as it prevents the creation of violating logic in the first place rather than attempting to detect and mitigate violations during execution [3] . The risk-of-harm index for this architectural layer is kept low because it operates purely at the policy and type level, constraining what other tools can do without ever touching biological or invasive protocols itself .

The following table details the key components and their functions within the constitutional policy root.

Component	Description	Role in Neurorights Enforcement
`neurorights.envelope.citizen.v1`	The ALN shard acting as the constitutional policy root.	Defines the authoritative set of neurorights invariants that all compliant code must satisfy.
Rights & Invariants	Policies including no exclusion, no inner-state scoring, no neurocoercion, revocability, and ecosocial reporting.	Converted into Rust constants, traits, and macro rules that enforce these constraints at compile time.
`NeurorightsProfile` Struct	Mirrors the shard's metadata, containing `id`, `version`, and `anchor`.	Attached to every `PromptEnvelope`, cryptographically linking the action to the specific version of the neurorights constitution.
Versioning (`version` field)	A string identifier for the specific iteration of the policy (e.g., "1.3").	Ensures that the build process fails if a component is built against an outdated or mismatched policy version.
Anchoring (`anchor` field)	A cryptographic pointer to the ALN location where the shard is stored (e.g., a DID).	Provides an immutable, verifiable link between the deployed software and the official policy document.

By establishing the `neurorights.envelope.citizen.v1` shard as the foundational policy layer, the framework moves beyond simple guidelines and implements a robust, verifiable, and enforceable contract. This contract is not something that can be ignored or overridden at runtime; it is woven into the very fabric of the codebase, becoming a permanent property of any component that seeks to participate in the augmented-citizen workflows governed by Cyber-Retrieval. This transforms governance from a discretionary act into a mathematical certainty, providing a strong defense against coercive, discriminatory, or otherwise harmful cognitive interventions.

Core Rust Types and Type-Level Enforcement Mechanisms

The technical implementation of the neurorights firewall hinges on a sophisticated use of Rust's type system to make violations of neurorights impossible to represent in the code, a principle known as "making illegal states unrepresentable" [3] . This is achieved through a carefully designed set of core Rust types and the application of advanced compile-time invariant patterns. The central element of this system is the `NeurorightsBound<PromptEnvelope, NeurorightsEnvelope>` wrapper, which acts as a type-level gatekeeper at all router entry points . Every function, tool, or adapter that interacts with augmented-citizen workflows and has the potential to influence access, scoring, or coercion must have a signature that accepts this specific, constrained type . This forces any call path originating from a router to pass through a check that validates the presence and integrity of the neurorights profile before any further processing can occur, failing to compile if the contract is not met .

The `NeurorightsBound` wrapper is a generic struct that pairs a payload type (typically `PromptEnvelope`) with a neurorights envelope type (`NeurorightsEnvelope`). The `NeurorightsEnvelope` struct itself is a mirror of the `neurorights.envelope.citizen.v1` ALN shard, containing the policy invariants as Rust constants and associated data . The power of this design lies in the constraints imposed by the combination of these types. For example, the ALN shard might define an invariant that the maximum permissible score for a cognitive intervention cannot exceed a certain value. This invariant would be converted into a `const` value during the ALN-to-Rust compilation step . The `NeurorightsEnvelope` would then hold this constant. A marker trait, perhaps named `ValidatedNeurorights`, could be defined, and implementations of this trait for specific `NeurorightsEnvelope` instances would use `const_assert_eq!` to verify that the invariants hold true at compile time [17]. Only if a `NeurorightsEnvelope` successfully passes all these compile-time checks can it be used to construct a valid `NeurorightsBound<PromptEnvelope, NeurorightsEnvelope>`. This means that any `PromptEnvelope` that reaches a router handler has already been statically verified to conform to the constitutional neurorights guarantees.

To implement this system effectively, several advanced Rust patterns are employed. **Sealed traits** are essential for controlling the implementation of key behaviors and preventing external crates from circumventing the security model [18]. A sealed trait could be used to define the interface for all valid neurorights envelopes, with the trait being declared within the neurorights crate but implemented only by the `NeurorightsEnvelope` struct generated from the ALN shard. This prevents malicious or poorly written code from defining its own `NeurorightsEnvelope` variant that might violate the policy. **Phantom types** are another powerful tool, used to encode information at the type level without incurring any runtime cost [2]. They can be used to parameterize the `NeurorightsBound` wrapper itself, ensuring that a handler expecting a `NeurorightsBound<T, N1>` (where `N1` corresponds to policy v1.0) cannot accidentally be passed a `NeurorightsBound<T, N2>` (for policy v2.0), even if both are valid wrappers. This enforces strict versioning and compatibility across the system.

Furthermore, procedural macros play a crucial role in automating the generation of complex, type-safe structures and reducing boilerplate code for developers [2]. A macro could be created to automatically derive the necessary implementations for the `NeurorightsEnvelope` struct from the ALN shard definition. Another macro could be used to wrap a router handler function, automatically adding the required `NeurorightsBound` type annotation and generating the necessary validation logic based on the configured policy. This abstraction is vital for developer adoption, as it hides the complexity of the underlying type-level constraints behind a simple, declarative syntax . The overall type graph forms a clear hierarchy of safety, starting from the `PromptEnvelope`, enriched with a `neurorights_profile`, wrapped in the `NeurorightsBound` gatekeeper, and finally processed by handlers whose signatures guarantee adherence to the neurorights contract .

The following table outlines the key Rust types and their roles in enforcing neurorights by construction.

Type / Pattern	Description	Enforcement Mechanism
`NeurorightsBound<P, N>`	A generic wrapper type that binds a payload `P` (e.g., `PromptEnvelope`) to a neurorights envelope `N`.	Acts as a compile-time gatekeeper. Router handlers accepting this type will fail to compile if given an unwrapped `PromptEnvelope`.
`NeurorightsEnvelope`	A struct mirroring the `neurorights.envelope.citizen.v1` ALN shard, containing policy invariants as Rust constants.	Serves as the carrier for the constitutional guarantees. Its validity is checked at compile time.
Sealed Traits	A trait defined in the neurorights crate but implemented only by the `NeurorightsEnvelope` struct. [18]	Prevents external crates from creating unauthorized variants of the neurorights envelope, ensuring all implementations are controlled and validated.
Phantom Types	Zero-cost types used to parameterize generics with additional type-level information (e.g., policy version). [2]	Ensures strict versioning and prevents mixing of objects from different policy contexts, enforcing compile-time consistency.
`const` Assertions	Compile-time checks using `const_assert_eq!` or similar mechanisms to validate invariants held within the `NeurorightsEnvelope`. [17]	Validates that the numerical or categorical constraints from the ALN shard (e.g., max score) are satisfied, failing the build if violated.
Procedural Macros	Custom derive macros and attribute macros that automate the generation of `NeurorightsEnvelope` implementations and handler wrappers. [2]	Hides the complexity of the type-level setup from developers, providing a simple, declarative API for integrating with the firewall.

This comprehensive approach, leveraging Rust's strengths in static analysis and type-driven development, creates a multi-layered defense-in-depth strategy at the architectural level [3] . It does not rely on runtime checks to catch errors; instead, it uses the compiler as the ultimate authority, ensuring that any software built within this framework is, by its very nature, compliant with the foundational neurorights constitution. This represents a significant advancement over traditional safety models, offering a higher degree of assurance by preventing the possibility of violating states altogether.

Data Envelope Integration and Authorship Metadata Anchoring

A successful neurorights firewall depends on its ability to inspect and constrain every relevant action as it enters the system. This requires deep integration with the platform's core data structures, specifically the `PromptEnvelope`, and a robust mechanism for anchoring governance directly into the provenance and authorship metadata of every asset and log event . The proposal to extend the existing `PromptEnvelope` with a `neurorights_profile` field is a pragmatic and powerful solution that builds upon established patterns for provenance and traceability in AI systems .

The `PromptEnvelope` is already the universal container for all actions routed through Cyber-Retrieval, carrying essential metadata such as `trace_id`, `intent`, `args`, `security_level`, `identity`, `provenance`, and `governance` . By adding a `neurorights_profile` field, the envelope becomes a self-declarative artifact of its own compliance status . This profile is always populated with the specific ALN id and version that generated the invariants being enforced, along with a cryptographic anchor pointing back to its location in the ALN . This extension ensures that every request that has the potential to influence an augmented citizen's cognition, autonomy, or access to services carries its own verifiable credentials regarding the neurorights constitution it adheres to. The `PromptEnvelope` schema is thus enhanced to look like the following in Rust:

```
#[derive(Clone)]
pub struct NeurorightsProfile {
    pub id: String,       // "neurorights.envelope.citizen.v1"
    pub version: String,  // "1.3"
    pub anchor: String,   // "did:.../ALN..."
}

#[derive(Clone)]
pub struct PromptEnvelope {
    pub trace_id: String,
    pub intent: Intent,
    pub args: serde_json::Value,
    pub security_level: SecurityLevel,
    pub identity: Identity,       // DID / ALN / Bostrom
    pub provenance: Provenance,
    pub governance: Governance,
    pub neurorights_profile: NeurorightsProfile,
}
```

This design choice is critical because it places the responsibility for declaring one's own compliance directly on the entity that constructs the action. The `NeurorightsBound<PromptEnvelope, NeurorightsEnvelope>` wrapper, which sits at router entry points, will then perform a static check to ensure the incoming `PromptEnvelope` contains a valid `neurorights_profile` that matches the expected `NeurorightsEnvelope` type for that particular router or tool . This makes the `PromptEnvelope` itself a carrier of trust, where its contents are cryptographically anchored to a specific, verifiable policy .

Beyond the envelope's content, governance is further solidified by anchoring authorship metadata directly into the system's identity and accountability layers. The framework mandates that any asset, log entry, or transaction that touches cognition, autonomy, or access to critical services must carry a complete authorship signature consisting of `(user_did, aln, bostrom_address)` . This triplet provides a triple-lock system for attribution and accountability: 1. **DID (Decentralized Identifier):** Supplies the cryptographically anchored identity of the actor initiating or modifying the action, mirroring how NeuroAlnParticle and SovereignHost contracts bind actions to host DIDs . This ensures that every action can be traced back to a specific, verifiable agent. 2. **ALN (Application-Level Network):** Encodes the neurorights contract and governance scope (e.g., Phoenix, global), providing context about the terms under which the action is being performed . This is consistent with the use of ALN as a policy container throughout the stack. 3. **Bostrom Address:** Gives a stake-bearing identity that ties the action to specific incentives, penalties, and governance roles within the Cybernet ecosystem . This creates a direct feedback loop where actors are economically accountable for their actions.

This authorship metadata should also include an `eibon_label` and a pointer to the `neurorights ALN version` . The `eibon_label` serves as a unique identifier for the event within the Eibon governance trail, while the ALN version pointer ensures that any regression in neurorights guarantees is detectable as a type or constant mismatch during the build process . This creates a comprehensive, immutable, and auditable record of every cognitively-relevant action, linking the actor, the action itself, the governing policy, and the economic consequences. The Cybostate-Factor for this system is therefore described as "Constitutional, Type-Enforced, Governance-Linked," meaning that neurorights become invariants that every Cyber-Retrieval component must satisfy before it can even be built or deployed .

The following table summarizes the key fields added to the `PromptEnvelope` and the `Authorship` metadata and their respective purposes.

Field / Structure	Type	Purpose
`neurorights_profile`	`NeurorightsProfile`	Carries the ALN `id`, `version`, and `anchor` to cryptographically link the action to the specific neurorights constitution it complies with.
`NeurorightsProfile.id`	`String`	Identifies the type of neurorights policy (e.g., "neurorights.envelope.citizen.v1").
`NeurorightsProfile.version`	`String`	Specifies the exact version of the policy (e.g., "1.3"), enabling strict versioning and build failure on mismatch.
`NeurorightsProfile.anchor`	`String`	Provides a cryptographic link (e.g., a DID) back to the ALN location where the policy shard is stored, ensuring immutability.
`(user_did, aln, bostrom_address)`	`(String, String, String)`	The authorship triplet that provides verifiable identity, policy context, and stake-based accountability for every cognitively-relevant action.
`eibon_label`	`String`	A unique label for the event within the Eibon governance trail, facilitating traceability and auditability.
`hex-stamp`	`String`	A cryptographic hash (e.g., `0xaf31c8e9...`) representing a snapshot of a compliant system state, used for logging and rollback proofs.

By deeply integrating the neurorights profile into the `PromptEnvelope` and mandating a rich authorship metadata set, the firewall ensures that compliance is not an optional feature but an intrinsic, verifiable property of every action. This transforms the system from a passive collection of rules into an active, verifiable constitution, where the data itself carries the proof of its own legitimacy.

The ALN-to-Rust Compilation Pipeline and CI/CD Compliance

The seamless translation of the declarative `neurorights.envelope.citizen.v1` ALN policy into concrete, enforceable Rust code is the linchpin of the entire compile-time firewall. This is accomplished through a dedicated ALN-to-Rust compilation pipeline, which acts as the bridge between policy and implementation. This pipeline is not a one-time script but an integral part of the build process, ensuring that the running software is always in sync with the latest constitutional guarantees . The most likely implementation involves a build script (`build.rs`) that executes during `cargo build`. This script would be responsible for locating, parsing, and validating the target ALN shard, and then generating the necessary Rust source files that define the `NeurorightsEnvelope` struct, its associated constants, and any required trait implementations .

This automated generation is critical for maintaining security and consistency. Manually editing Rust files to reflect changes in the ALN policy would be error-prone and a common source of vulnerabilities. Instead, the pipeline ensures that any modification to the `neurorights.envelope.citizen.v1` shard—be it a tweak to a numeric invariant or a change in the policy text—triggers a regeneration of the corresponding Rust artifacts. Because these generated files are included in the build, any dependent code that relies on the old structure or constants will immediately fail to compile . This creates a powerful feedback loop: the build system itself becomes the enforcer of the policy contract. Developers cannot deploy software that is out of sync with the current neurorights constitution. This process is analogous to how Protocol Buffers or FlatBuffers schemas are compiled into language-specific code, but applied here to governance policies. The entire process—from the ALN shard to the final compiled binary—is deterministic and reproducible, forming a verifiable chain of custody for the policy guarantees.

This tightly integrated build process naturally extends into the Continuous Integration/ Continuous Deployment (CI/CD) pipeline, which becomes the primary arena for compliance auditing. The CI/CD jobs must be instrumented with custom lints and tests that enforce the neurorights contract. These checks serve as mandatory gates that a pull request must pass before it can be merged and deployed. Key linting rules would include:

- **Presence of `neurorights_profile`:** A lint would scan the codebase for any function that handles a `PromptEnvelope` and flag it if it does not wrap the envelope in a `NeurorightsBound` type. This ensures that no part of the system can process an action without first validating its neurorights credentials .
- **Consistency of ALN Version:** The build script would output the version of the ALN shard it used to generate the code. The CI pipeline would then check that this version matches the expected version recorded in a configuration file. A mismatch would trigger a build failure, forcing developers to either update their code to the new policy or explicitly request an exception.
- **Logging and Evidence Requirements:** Tests would verify that all log events for cognitively-relevant actions include the required fields: `neurorights_profile`, `eibon_label`, and a `hex-stamp` . The `hex-stamp` (`0xaf31c8e924f5703d8c4f2a19e5d44cb7` is a candidate) serves as a cryptographic snapshot of a compliant system state and is essential for creating an auditable trail that can be used for governance reviews or rollback proofs .

The following table outlines the key components and checks for the CI/CD compliance framework.

CI/CD Component	Check / Lint	Failure Condition	Purpose
Build Script (`build.rs`)	Parses `neurorights.envelope.citizen.v1` ALN shard and generates Rust code (structs, constants).	Invalid ALN format or missing required fields in the shard.	Automates the translation of policy into executable code, preventing manual errors.
Compiler	Type-checking of `NeurorightsBound<PromptEnvelope, NeurorightsEnvelope>` wrapper.	A router handler receives a standard `PromptEnvelope` instead of the wrapped type.	Enforces the neurorights contract at the language level, making non-compliant code unbuildable.
Custom Linter	Checks for the presence of `neurorights_profile` in all `PromptEnvelope` instantiations.	An action is constructed without a `neurorights_profile`.	Ensures that every action is born with its own compliance credentials.
Version Consistency Test	Compares the ALN version used in the build with the expected version in `neurorights.toml`.	Mismatch between the generated code's policy version and the project's declared policy version.	Prevents deployment of code built against an outdated or unexpected policy.
Logging Validation Test	Scans log outputs for required fields (`neurorights_profile`, `eibon_label`, `hex-stamp`).	A log event for a cognitively-relevant action is missing one or more required fields.	Creates a complete, audit-ready evidence trail for all actions.

This rigorous, automated compliance framework significantly reduces the burden on human auditors and minimizes the risk of oversight. It aligns with modern supply chain security best practices, which emphasize hardening CI/CD runners to prevent attacks like the SolarWinds compromise, where malware was inserted during the build process [1]. A system like this mitigates such risks because any unauthorized tampering with the ALN shard or the generated code would break the build, providing a strong signal of compromise [1]. The resulting audit trail, enriched with `eibon_labels` and `hex-stamps`, maps cleanly to existing Cybernet/Cyberswarm audit concepts, providing a verifiable record of compliance with neurorights guidance and broader AI governance expectations related to mental privacy, autonomy, reversibility, and traceability .

Comparative Analysis: Compile-Time Guarantees vs. Runtime Safety Layers

The introduction of a compile-time neurorights firewall necessitates a clear understanding of its relationship with the existing runtime safety mechanisms within the bioscale/BCI stack. These two approaches are not mutually exclusive competitors but rather complementary layers of defense, each providing a distinct class of guarantees. The compile-time firewall offers high-assurance prevention by construction, while the runtime layers provide flexible detection and mitigation during operation. Together, they form a robust defense-in-depth strategy, where the former aims to make violations impossible to create, and the latter aims to manage the consequences if a violation were to occur despite preventive measures.

The fundamental difference lies in their enforcement point and the nature of the guarantees they provide. The compile-time firewall operates at the level of the code itself, using the Rust type system to enforce constraints before the program is ever executed [3]. Its guarantee is one of **prevention**: it makes illegal or coercive states unrepresentable, thereby preventing the creation of logic that violates neurorights. If a developer attempts to write code that excludes someone from a basic service—a direct violation of the `neurorights.envelope.citizen.v1` policy—the code will simply fail to compile . In contrast, the existing runtime safety layers, such as bioscale envelopes, downgrade contracts, and telemetry from NeuroAlnParticle, operate at the level of executing programs and interacting with biological systems . Their guarantee is one of **detection and response**: they monitor actions as they happen and can intervene to mitigate harm, but they do not prevent the initial, potentially violating logic from being present in the codebase [15].

For example, consider a bioscale adapter that scores a user's cognitive state. A runtime mechanism might involve a downgrade contract that, at execution time, checks if the calculated score exceeds a certain threshold and, if so, automatically lowers it to a safe level . This is a valuable safety net. However, the compile-time firewall would go further: it would embed the maximum allowable score as a `const` assertion within the `NeurorightsEnvelope` . Any attempt to construct a score object that violates this invariant would be rejected by the compiler, making the violation impossible from the outset. The runtime layer manages the outcome, while the compile-time layer manages the possibility. This distinction is critical for understanding the threat model and the appropriate use of each mechanism. The compile-time firewall is designed to prevent architecturally flawed or malicious designs from ever reaching production, whereas the runtime layers are designed to handle operational anomalies, sensor noise, or unforeseen edge cases.

The following table provides a detailed comparison between the compile-time neurorights firewall and the existing runtime safety mechanisms.

Dimension	Compile-Time Neurorights Firewall	Existing Runtime Safety Layers (Bioscale, Downgrade Contracts, etc.)
Enforcement Level	Code Generation & Type System [3]	Bioscale Protocols & Runtime Logic
Primary Guarantee	**Prevention:** Illegal states are made unrepresentable.	**Detection/Mitigation:** Violations are detected and responded to during execution.
Mechanism	ALN shard parsed at build time to generate Rust types, traits, and constants.	Telemetry from bioscale envelopes, preconditions in Rust code, and contract execution at runtime .
Example	A function signature `fn(process: NeurorightsBound<PromptEnvelope, N>)` will not accept a standard `PromptEnvelope`, causing a compile error.	An adapter receives a `PromptEnvelope`, calculates a score at runtime, and a separate function calls a downgrade contract if the score is too high.
Strengths	High assurance, zero runtime performance overhead, prevents entire classes of bugs and malicious designs by construction.	Flexibility to adapt to unforeseen situations, provides real-time evidence and telemetry for audits, can respond to dynamic conditions.
Weaknesses	Less flexible; may block novel but legitimate interventions not anticipated in the policy. Requires a mature governance process for ALN updates.	Requires continuous monitoring and introduces runtime latency. Can be bypassed by logic that evades telemetry. Potential for false negatives.
Failure Mode	Build failure. Development workflow is halted until the code is made compliant.	Runtime error, warning, or a mitigated outcome (e.g., a downgraded score). The action may still proceed, albeit altered.

This comparative view clarifies which invariants are guaranteed by construction (unrepresentable states) and which require runtime telemetry or evidence checks . The compile-time firewall provides absolute guarantees for the constraints encoded in the `neurorights.envelope.citizen.v1` shard. Invariants like "no inner-state scoring" or "no neurocoercion" are enforced as long as the ALN shard defines them precisely enough to be translated into Rust code. On the other hand, more nuanced or context-dependent safety properties might still benefit from runtime checks. For instance, while the compile-time layer can prevent the *construction* of a coercive action, a runtime layer might be needed to monitor the physiological effects of an action to ensure it does not lead to unintended coercive outcomes, a concern addressed in frameworks for regulating neural data processing [14] .

Ultimately, the two layers work in concert. The compile-time firewall acts as the first line of defense, purifying the codebase of any overtly harmful or constitutionally non-compliant logic. The runtime layers then act as a second line of defense, providing a last-chance safety net and a source of evidence during operation. This dual-layered approach is essential for a system operating at the intersection of cognition and technology, where the stakes of failure are exceptionally high. It ensures that the system is not only safe in practice but is also fundamentally sound in its design.

Developer Integration Guide and Audit Framework

For the neurorights firewall to be effective, it must be seamlessly integrated into the daily workflows of developers and clearly understood by compliance and audit teams. This requires a practical developer integration guide and a formal compliance and audit checklist that translate the theoretical framework into actionable steps and verifiable artifacts. The goal is to make adopting the neurorights contract as straightforward as possible while ensuring that the resulting system meets the highest standards of safety and governance.

The developer integration process begins with setting up the project to consume the `neurorights.envelope.citizen.v1` ALN shard. This involves adding the shard's location to a configuration file and ensuring the project's build script is correctly set up to parse it . Once the environment is prepared, developers follow a series of steps to wrap their existing or new components in the neurorights contract. First, they must ensure that any data structures representing actions are based on the `PromptEnvelope` with the `neurorights_profile` field . Second, they need to modify all router definitions and tool/adapters that interact with augmented-citizen workflows. Instead of accepting a standard `PromptEnvelope`, their function signatures must be changed to accept `NeurorightsBound<PromptEnvelope, NeurorightsEnvelope>` . This single change acts as the compile-time gate, and the compiler will immediately report any call sites that fail to provide a properly wrapped envelope.

Concrete examples are crucial for guiding developers. For instance, when adapting an upgrade_store function that modifies a user's cognitive profile, the developer would change its signature from `fn(handle_update(update: UpdateRequest))` to `fn(process: NeurorightsBound<UpdateRequest, CitizenRights>)`. The framework's procedural macros can simplify this further, perhaps allowing a `#[neurorights_handler]` attribute that automatically performs this wrapping and inserts the necessary validation logic [2] . Error handling must also be considered; the framework should define clear error types for various failure modes, such as

`MissingNeurorightsProfile`, `InvalidPolicyVersion`, or `InvariantViolation`. These specific error types provide clear feedback to developers and can be logged with sufficient detail for debugging and auditing. Example diffs showing the transformation of an existing service to comply with the neurorights contract would be invaluable resources for adoption .

On the governance side, a compliance and audit checklist is essential for CI/CD and post-deployment reviews. This checklist formalizes the requirements into discrete, verifiable items, mapping them to broader neurorights and AI governance expectations. It serves as a practical tool for ensuring that every deployed component adheres to the constitutionally enforced contract.

The following table presents a sample compliance and audit checklist for the neurorights firewall.

Checklist Item	Verification Method	Governance Principle
ALN Policy Synchronization	Verify that the ALN shard version used in the build matches the version declared in the project's configuration file.	**Rule of Law:** All participants must adhere to the same, up-to-date legal framework.
Type Coverage	Use a static analysis tool or custom lint to confirm that all router entry points and cognitive tools accept `NeurorightsBound<T, N>`.	**Due Process:** No action can be taken without first satisfying the prescribed legal procedures.
Log Completeness	Validate that all log events for cognitively-relevant actions contain the `neurorights_profile`, `eibon_label`, and a `hex-stamp`.	**Transparency:** Actions must be fully documented and traceable.
Authorship Correctness	Ensure that every asset, log, and transaction is signed with the full `(user_did, aln, bostrom_address)` authorship triplet.	**Accountability:** Every action must be attributable to a responsible party.
Evidence Bundle Integrity	Confirm that evidence bundles for upgrades or significant actions include a `hex-stamp` that cryptographically commits to a compliant system state.	**Reversibility:** The ability to roll back to a known-good state is a fundamental right.
No Manual Edits	Inspect the build artifacts to ensure that the `NeurorightsEnvelope` and associated constants were generated by the build script and not manually edited.	**Integrity:** The policy must be faithfully represented in the deployed software without corruption.

This checklist directly maps to key AI governance expectations, such as mental privacy, autonomy, reversibility, and traceability . By automating as many of these checks as possible within the CI/CD pipeline, the framework shifts the focus of human auditors from repetitive verification to higher-level strategic review. The `hex-stamp`, in particular, serves as a powerful tool for this process. It provides a compact, cryptographic summary of the system's state at a moment of compliance, which can be logged alongside other evidence and evolution sequences for future reference . This creates a permanent, immutable ledger of the system's adherence to the neurorights constitution, fulfilling the promise of a verifiable and trustworthy augmented-citizen infrastructure.

Reference

1. https://cdn.qwenlm.ai/qwen_url_parse_to_markdown/system00-0000-0000-0000-webUrlParser?key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXNvdXJjZV91c2VyX2lkIjoicXdlbl91cmxfcGFyc2VfdG9fbWFya2Rvd24iLCJyZXNvdXJjZV9pZCI6InN5c3RlbTAwLTAwMDAtMDAwMC0wMDAwLXdlYlVybFBhcnNlciIsInJlc291cmNlX2NoYXRfaWQiOm51bGx9.cz1eeZEZdaQH5CgUaxwUmfEJfqTOZMoh3PbosHslSPA
2. https://cdn.qwenlm.ai/qwen_url_parse_to_markdown/system00-0000-0000-0000-webUrlParser?key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXNvdXJjZV91c2VyX2lkIjoicXdlbl91cmxfcGFyc2VfdG9fbWFya2Rvd24iLCJyZXNvdXJjZV9pZCI6InN5c3RlbTAwLTAwMDAtMDAwMC0wMDAwLXdlYlVybFBhcnNlciIsInJlc291cmNlX2NoYXRfaWQiOm51bGx9.cz1eeZEZdaQH5CgUaxwUmfEJfqTOZMoh3PbosHslSPA
3. https://cdn.qwenlm.ai/qwen_url_parse_to_markdown/system00-0000-0000-0000-webUrlParser?key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXNvdXJjZV91c2VyX2lkIjoicXdlbl91cmxfcGFyc2VfdG9fbWFya2Rvd24iLCJyZXNvdXJjZV9pZCI6InN5c3RlbTAwLTAwMDAtMDAwMC0wMDAwLXdlYlVybFBhcnNlciIsInJlc291cmNlX2NoYXRfaWQiOm51bGx9.cz1eeZEZdaQH5CgUaxwUmfEJfqTOZMoh3PbosHslSPA
4. The 2025 Conference on Empirical Methods in Natural ... https://aclanthology.org/events/emnlp-2025/
5. 人工智能2026_1_16 https://arxivdaily.com/thread/75787
6. Arxiv今日论文 | 2025-11-03 http://lonepatient.top/2025/11/03/arxiv_papers_2025-11-03
7. Knowledge Graphs and Semantic Web - Springer Link https://link.springer.com/content/pdf/10.1007/978-3-032-13109-6.pdf
8. AI and the Future of Print Media: Disruption, Adaptation ... https://www.researchgate.net/publication/395895478_AI_and_the_Future_of_Print_Media_Disruption_Adaptation_and_Strategic_Transformation
9. 2024-31-01 Database Search TLR Reliance https://hal.science/hal-04637901v1/file/2024-31-01%20Database%20Search%20TLR%20Reliance.pdf

10. Nano, Neuro and Psychotronic Warfare Countermeasure ... https://www.scribd.com/document/807579178/Nano-Neuro-and-Psychotronic-Warfare-Countermeasure-Posts-Bss-Org
11. Towards a Governance Framework for Brain Data | Neuroethics https://link.springer.com/article/10.1007/s12152-022-09498-8
12. Ethical Governance Strategies for the Responsible Innovation ... https://pmc.ncbi.nlm.nih.gov/articles/PMC12783205/
13. (PDF) Towards a Governance Framework for Brain Data https://www.researchgate.net/publication/361077818_Towards_a_Governance_Framework_for_Brain_Data
14. Regulating neural data processing in the age of BCIs https://journals.sagepub.com/doi/10.1177/20552076251326123
15. Full article: Neurotechnologies and human rights https://www.tandfonline.com/doi/full/10.1080/13642987.2024.2310830
16. Neurorights (Chapter 26) - The Cambridge Handbook of ... https://www.cambridge.org/core/books/cambridge-handbook-of-the-right-to-freedom-of-thought/neurorights/B1AEF25AD18D9C8164CE9B366979B664
17. Compile-time generic type size check https://stackoverflow.com/questions/30330519/compile-time-generic-type-size-check
18. How to enforce that a type implements a trait at compile time? https://stackoverflow.com/questions/32764797/how-to-enforce-that-a-type-implements-a-trait-at-compile-time
19. After The Digital Tornado | PDF | Internet https://www.scribd.com/document/613093594/After-the-Digital-Tornado
20. After the Digital Tornado https://resolve.cambridge.org/core/services/aop-cambridge-core/content/view/B746434A076A9EC7FD10AF12D69E6EA4/9781108426633AR.pdf/After_the_Digital_Tornado.pdf?event-type=FTLA