From Specification to Guarantee: An ALN-to-Rust Workflow for Unrepresentable Unsafe States and Audit-Ready Neurorights

This report provides a comprehensive deep-dive analysis into the refinement of an Application Layer Notation (ALN) to Rust code generation pipeline. The central objective is to establish a robust, auditable, and safety-first framework for bioscale-compliant systems. The analysis synthesizes user directives and contextual materials to construct a detailed strategy focused on treating ALN as a formal type system, architecting a canonical layer for transhuman rights, integrating metrics without compromising safety, and enforcing compliance through a rigorous CI/manifest validation process. The ultimate goal is to achieve "audit-readiness by construction," where governance, traceability, and safety are inherent properties of the generated artifacts rather than afterthoughts. The research is structured around a phased approach, beginning with foundational safety and progressing to layered features that enhance observability and accountability.

Foundational Principle: Hardening Safety and Type Enforcement

The cornerstone of the refined ALN-to-Rust workflow is the elevation of Application Layer Notation (ALN) from a simple declarative configuration format to a formal type system [36] [46]. This principle dictates the entire architecture, shifting the paradigm from runtime checks to compile-time guarantees. The core directive is to generate Rust code where unsafe states and prohibited actions are literally unrepresentable in safe code . This approach leverages Rust's powerful type system and ownership model to eliminate entire classes of vulnerabilities before execution begins, creating a foundation of trust upon which all other system properties are built [47] [65]. This section details the critical components required to realize this foundational principle, focusing on the development of a static ALN validator, the generation of prohibitive guards, and the enforcement of monotonic safety invariants.

A pivotal step in this process is the creation of a dedicated ALN validator tool. This tool must operate as a pre-compile phase, statically analyzing `.aln` files to enforce a strict set of rules derived directly from the safety policies defined within them . Its primary function is to reject any schema containing explicitly forbidden fields that could lead to unsafe actuation. Examples of such fields include `torque`, `current`, `stimpattern`, and `autopilot` . By identifying and rejecting these schemas early in the development lifecycle, the validator prevents their corresponding unsafe functionality from ever being represented in the generated Rust crate. This is a significant improvement over runtime checks, which rely on correct implementation and execution; if the check is bypassed, skipped, or contains a bug, the system remains vulnerable. The validator acts as the first line of defense, ensuring the integrity of the source specification before it is even processed by the code generator. Procedural macros in Rust provide a suitable mechanism for implementing such a tool, as they can emit compilation errors directly back to the developer, halting the build process immediately upon encountering a violation [60] .

Beyond prohibiting specific fields, the validator must also enforce complex safety constraints encoded as monotonic inequalities. These inequalities, such as $G_{\text{new}} \le G_{\text{old}}$, $D_{\text{new}} \le D_{\text{old}}$, and $R_{\text{new}} \le R_{\text{old}}$, are critical for ensuring that updates to software, OTAs, or policies cannot relax existing safety envelopes . For instance, a constraint might stipulate that the maximum permissible torque (`G`) or duty cycle (`D`) must not increase from one version of a control algorithm to the next. The ALN validator must parse these constraints and programmatically verify them against the values specified in the new schema relative to a known baseline. If a proposed update would violate a monotonic constraint—for example, by setting a new `max_torque_nm` value higher than the previous one—the validator must reject the change . This capability is essential for maintaining a conservative safety posture, especially in systems subject to frequent updates. The enforcement of such mathematical invariants at the ALN level ensures that the resulting Rust types and logic inherently respect these bounds, preventing representable paths to less-safe operational modes [38] .

Once a valid ALN schema has been approved by the validator, the next stage is to translate its specifications into strongly-typed Rust code that enforces these policies. This is achieved through a sophisticated procedural macro-based code generation engine. The goal is to produce Rust "guard crates" that wrap all device access and expose only a safe, well-defined API . The code generator consumes the validated ALN shard and produces Rust modules containing structs and enums that mirror the ALN's structure. For example, a session state defined in ALN as `Uninitialized → Calibrating → Ready → Streaming → Faulted` would be translated into a Rust `enum BciSessionState` . Crucially, the code generation process must be intelligent enough to omit functions from the public API that correspond to prohibited actions. If an ALN policy stanza specifies `stimulation_allowed false`, the generated `EegDeviceGuard` struct must contain no public method named `start_stimulation` or equivalent . This directly implements the principle of making unsafe constructs unrepresentable. Developers interacting with the generated crate would find that attempting to call a non-existent function for stimulation would result in a compile-time error, providing a stronger guarantee than a runtime check that might simply return an error code. This pattern is analogous to using sealed traits in Rust to limit the implementations of a trait to a specific module, thereby controlling the universe of possible behaviors [27] . The code generator effectively creates a bespoke, policy-aware abstraction layer tailored to each unique ALN specification.

This type-level enforcement extends to all aspects of the ALN specification, including device descriptors, safety policies, and metric requirements. Each block in the ALN file—such as `bci.device`, `biomech.device`, and `metric`—is transformed into corresponding Rust structs and types. For example, the `bci.device EEG-Headcap-32` stanza would generate a struct defining the properties of that device, while the `policy safety` block would inform the logic within the guard implementations . The combination of a static ALN validator and a context-aware code generator forms a powerful compiler that translates high-level safety contracts into low-level, provably-safe executable code. This approach aligns with best practices in safety-critical software development, which emphasize determinism, predictability, and end-to-end traceability from requirements to implementation [53] . By building this capability into the development workflow, the system becomes inherently more secure-by-design, reducing the cognitive load on developers and minimizing the potential for human error when implementing safety logic manually [51] . The focus on creating a robust ALN-to-Rust type system is therefore not merely an optimization but the fundamental prerequisite for achieving the project's ambitious goals of safety, auditability, and rights preservation.

Architectural Innovation: Canonical Neurorights Envelopes

A key architectural innovation within the refined ALN-to-Rust workflow is the strategic separation of transhuman rights fields from individual policy stanzas. Instead of embedding rights such as `noscore_from_inner_state` and `noexclusion_basic_services` directly into every ALN particle or manifest, these fields are consolidated into a single, canonical neurorights envelope shard, such as `neurorights.envelope.citizen.v1` . Other ALN stanzas then reference this shard rather than duplicating its contents. This design pattern promotes consistency, reusability, and a single source of truth for fundamental rights, transforming abstract legal concepts into concrete, machine-checkable data structures that can be validated by the compiler, runtime, and CI tools . This section explores the rationale behind this architecture, its implementation in both ALN and Rust, and its profound implications for governance and formal verification.

The primary benefit of this reference-based approach is the establishment of a single, authoritative definition for core transhuman rights . When these fields are duplicated across multiple policy stanzas—for example, in payment, upgrade, and civic access policies—a risk emerges where the definitions could diverge due to human error or oversight. A change to `noexclusion_basic_services` in one policy might be missed in another, leading to inconsistent enforcement and potential violations of the intended rights. By defining the rights once in a dedicated ALN particle, such as `qpudatashards/particles/TranshumanRightsPhoenix2026v1.aln`, the system ensures that all downstream consumers are aligned with the same canonical contract . This shard becomes the ground truth, and any deviation from its prescribed values can be caught by the ALN validator or CI linter, enforcing uniformity across the entire system . This directly addresses the challenge of managing complex, cross-cutting concerns in a distributed system, ensuring that fundamental principles are not lost in translation between different domains.

From an implementation perspective, this architecture requires extensions to the ALN syntax and the code generation toolchain. The ALN syntax must support a `ref` keyword to allow stanzas to import the contents of another particle or shard . For instance, a payment policy stanza could declare `neurorightsenvelope ref neurorights.envelope.citizen.v1`. The code generator must then be capable of resolving these references at compile time, merging the contents of the referenced envelope into the local scope of the generating crate. In Rust, this concept is mirrored by the `TranshumanRightsEnvelope` struct defined in a dedicated crate, such as `crates/bioscale-governance/src/transhuman_rights.rs` . This struct serves as the direct representation of the ALN envelope, with fields like `pub noscore_from_inner_state: bool` and `pub noexclusion_basic_services: bool` . The procedural macros responsible for code generation will use libraries like `serde` to deserialize the JSON representation of the ALN envelope into this Rust struct

[62] [64]. This deserialized struct is then made available to the generated guards, allowing them to programmatically check the citizen's rights status before authorizing any action.

This architectural choice has significant implications for governance and formal verification. With neurorights encapsulated in a single, well-defined structure, they become a prime target for automated testing and formal proofs. For example, the `EqualityPaymentGuard` can be designed to specifically inspect the `noexclusion_basic_services` field of the referenced envelope. A suite of property tests can then be written to prove that, under all valid conditions where the user's balance is sufficient and the neurorights envelope indicates `noexclusion_basic_services: true`, the guard will always approve the transaction . While full automated proof frameworks for Rust are still emerging, the combination of generated guards, strict enums, and property tests provides a strong baseline for verifying these critical invariants [7] [48]. This transforms a policy promise into a computational guarantee. Similarly, the `cooperation_neurosafe_only` constraint can be formally checked to ensure that no law enforcement interaction results in coercive access to neural data when the citizen's envelope specifies this right . Tools like Creusot or Verus, which are designed for the deductive verification of Rust programs, could potentially be used to provide mathematical proof of correctness for these guard implementations [37] [49].

Furthermore, this pattern simplifies the policy stanzas themselves, making them more readable and focused on their specific domain logic. A payment policy no longer needs to be cluttered with declarations about mental privacy or basic service access; it can simply assert its dependency on a verified rights envelope. This modularity is a classic software engineering best practice, and its application here elevates the quality of the system's specifications. The table below illustrates the contrast between the duplicated and referenced approaches.

Feature	Duplicated Rights Approach	Referenced Envelope Approach

ALN Structure	`policy payment`
	`...`
	`neurorights.noscore_from_inner_state true`
	`neurorights.noexclusion_basic_services true`
		`policy payment`
		`...`
		`neurorightsenvelope ref`
		`neurorights.envelope.citizen.v1`

Rust Representation	`pub struct PaymentPolicy { pub noscore_from_inner_state: bool, pub noexclusion_basic_services: bool, ... }`	`pub struct PaymentPolicy { pub neurorights: TranshumanRightsEnvelope, ... }`
Enforcement Logic	Guard checks booleans directly from its own config struct.	Guard accesses the `neurorights` field of its config struct and calls methods on the `TranshumanRightsEnvelope` instance.
Consistency Risk	High. Requires manual synchronization across all policy stanzas.	Low. Relies on a single source of truth for the rights definition.
Testability	More complex. Tests must account for the possibility of divergent values.	Simpler. Tests can validate the behavior of the guard against a single, canonical envelope definition.

In summary, anchoring transhuman rights fields in a shared, referenceable invariant layer is a powerful architectural decision. It enhances consistency, reduces complexity, and enables deeper levels of automated verification and governance. By treating rights as a distinct, reusable data asset, the system moves closer to realizing the vision of a truly rights-aware and auditable infrastructure.

Metrics Integration and CI/Manifest Validation

After establishing a hardened safety and type enforcement layer, the workflow proceeds to two subsequent phases: integrating bounded eco-civic and performance metrics, and implementing a mandatory CI/manifest validation gate. These layers enrich the system with observability and accountability without compromising the foundational safety guarantees. Metrics integration allows for the real-time monitoring of both system performance and broader societal impact, while the CI/manifest layer operationalizes the enforcement of ALN contracts, ensuring that the generated artifacts conform to the established rules before deployment. This section details the strategies for wiring metrics into the generated code and designing a CI pipeline that acts as an immutable gatekeeper for invariant compliance.

The second priority in the workflow is the integration of seven canonical metrics and EcoHelp vectors into the generated Rust structs . These metrics, such as `DecoderAccuracy`, `EcoImpactScore`, and `AvgDailyDeviceHoursReduced`, provide quantitative signals for system health and social contribution . The critical constraint is that this integration must occur without reopening safety surfaces or introducing new points of failure . To achieve this, the system employs bounded wrapper types, such as those found in a `bioscale-metrics` crate, to constrain metric values to a specific range, for example, 0.0 to 1.0 . This `Bounded01` wrapper ensures that even if a calculation goes awry, the recorded metric value remains within a mathematically valid domain, preventing issues like negative energy consumption or accuracy scores greater than 100%.

The code generation process is responsible for creating the necessary telemetry infrastructure. For each metric stanza defined in the ALN file, the generator produces a corresponding entry in a `metrics.rs` module within the output crate . Using the `prometheus` crate, the generator creates `static Lazy` instances for counters, gauges, and histograms, namespaced with the study ID or ALN filename for clarity . For example, a required gauge like `bci_decoding_accuracy` would result in a generated static variable `pub static BCI_DECODING_ACCURACY: Lazy<Gauge> = Lazy::new(|| { ... })` . These metrics are then attached to the logic within the generated guards. A method inside the `EegDeviceGuard` might be responsible for updating the decoding accuracy gauge whenever a new epoch is processed, calling a generated helper function like `BCI_DECODING_ACCURACY.set(acc)` . This tight coupling ensures that telemetry is automatically collected wherever the guarded hardware or logic is used, providing a continuous stream of data for observability stacks and downstream orchestration systems .

The third and final layer of the workflow is the mandatory CI/manifest validation gate. This component ensures that the promises made in the ALN specification are upheld in practice. Every run of the CI pipeline must regenerate the Rust code from the ALN shards and then execute a series of validation jobs . A key artifact of this process is a machine-readable research manifest, typically a JSON file, that captures the complete context of the build . This manifest encodes crucial information such as the `study_id`, the path to the `aln_source` file, the name and version of the generated `crate`, the `git_sha` of the commit, a list of configured `devices`, registered `metrics`, and the effective `policies` . This manifest serves as an immutable, self-contained record of the system's configuration at a point in time.

The CI pipeline uses this manifest to perform several critical checks. First, it runs standard Rust tooling like `cargo fmt`, `cargo clippy`, and `cargo test` to ensure code quality and functional correctness . Second, and more importantly, it runs a manifest linter that validates the content of the generated JSON against the rules established by the ALN validator . This linter enforces the same checks performed during code generation: it verifies that no forbidden actuation fields were present in the source

ALN, that all monotonic safety inequalities were respected, and that all metrics fall within their specified ranges . Any pull request that fails this linting step is rejected by the CI gate, preventing non-compliant configurations from ever reaching the main branch or being deployed to production . This creates a closed-loop feedback system where changes to the ALN are continuously tested against the system's safety and policy constraints. The manifest itself becomes a trusted artifact, providing end-to-end traceability from the original ALN specification to the final deployed binary, a practice essential for achieving audit-readiness in safety-critical domains [53] .

The table below outlines the key components and their roles within this integrated workflow.

Component	Role in Workflow	Key Technologies / Concepts
ALN Metric Stanzas	Define required metrics and their types (gauge, histogram, counter).	`metric required bci_decoding_accuracy`
Code Generator	Produces `metrics.rs` with Prometheus `static Lazy` registrations.	Procedural Macros (`quote`, `syn`), `prometheus` crate [62] [63]
Rust Guards	Call metric setter functions (e.g., `.set()`) upon relevant events.	Generated `impl` blocks within guard structs
Research Manifest	A JSON artifact capturing the full build context: `study_id`, `aln_source`, `crate`, `devices`, `metrics`, `policies`.	Custom serializable Rust structs, `serde`
CI Pipeline	Regenerates code, runs tests, and lints the manifest.	GitHub Actions, GitLab CI, custom validation scripts
Manifest Linter	Validates manifest content against ALN rules (no forbidden fields, monotone safety, metric ranges).	Custom CLI tool, ALN schema definitions

By layering metrics integration and CI validation on top of the hardened safety foundation, the workflow achieves a holistic approach to system design. It provides not only the safety guarantees needed for reliable operation but also the observability and governance required for accountability and trust. This ensures that the system is not just safe, but demonstrably so, with every decision and parameter change leaving a verifiable, auditable trail.

Technical Implementation and Governance Co-development

The successful refinement of the ALN-to-Rust workflow hinges on closing critical technical implementation gaps while simultaneously co-developing governance and auditability features. The research directive emphasizes a pragmatic approach: prioritize work that is "audit-ready by construction," where governance is a thin layer of interpretation over rigorously structured, machine-verifiable artifacts . This involves finalizing the code generation engine, ensuring comprehensive guard coverage, developing specialized tooling for safety validation, and building out features like hex-traced decision logs and proofs of non-exclusion. This section delves into the specific technical tasks and governance features that must be developed in tandem to realize the project's objectives.

A primary technical gap to address is achieving full code generation completeness and guard coverage . The core of the system is a procedural macro-driven code generator that translates ALN schemas into Rust code. This requires the development of powerful macros like `alnbind!` and `metricderive!` to handle the complex mapping from ALN's declarative syntax to Rust's structured types and traits . This task is supported by ongoing research into type-constrained code generation with language models, which demonstrates that guiding LLMs with formal type system rules can significantly reduce compilation errors and improve functional correctness [10] [30] [36]. The generator must be capable of producing not only the basic device guard structs but also the complete hierarchy of neurorights envelopes and equality-credit ledgers . Once the code is generated, it is imperative to conduct a thorough audit of the system to ensure that *all* execution paths for sensitive operations—including payments, upgrades, and scheduling —flow exclusively through these newly created guards . Any direct access to hardware abstraction layers (HALs) or unguarded logic must be identified and wrapped, a process that can be aided by static analysis tools and adherence to a risk-based software engineering approach [51] [53].

Another critical area of development is tooling for monotone Over-the-Air (OTA) and safety-chain validation. Beyond the initial static ALN validator, a more sophisticated toolchain is needed to provide stronger assurances about system evolution. This can involve augmenting the validation pipeline with property-based tests using libraries like `proptest` [51]. Such tests can randomly generate sequences of policy updates and OTAs to probe for violations of safety invariants. Where feasible, this testing should be supplemented with formal verification techniques. Integrating tools like Creusot, Kani, or Prusti could allow for the mathematical proof of critical safety claims, such as muscle safety envelopes or eco-constraints, moving beyond statistical confidence to absolute certainty [7] [37] [48]. This advanced tooling represents the frontier of safety-critical software development and would significantly elevate the security posture of the system [5].

While closing these technical gaps, governance and auditability features must be co-developed to ensure the system is transparent and accountable. A key feature is the implementation of hex-traced decision logs. The `DecisionGuardOutcome` struct, which already includes a `hex_trace` field, is the foundation for this capability . This feature must be fully implemented and standardized across the system. Every AI gate that alters its behavior—denying a service, throttling a capability, or adjusting a score—must emit a log entry that includes this trace. The trace serves as a cryptographic or deterministic proof of the checks that were performed, allowing an external auditor to reconstruct the reasoning behind any decision without needing access to raw, sensitive neural data . This directly supports the right to explanation and contestability. Grounded proofs, represented by the hex strings, provide the evidentiary basis for these claims, linking abstract rights to concrete, verifiable outcomes .

Another crucial governance component is the formalization of proofs for non-exclusion. The `EqualityPaymentGuard` is highlighted as a prime candidate for this work . The research goal is to move beyond informal promises and develop formal proofs that demonstrate the guard's correctness. Specifically, one would aim to prove that the guard will *always* approve a transaction for a basic service whenever the user's financial balance is sufficient and their referenced neurorights envelope specifies `noexclusion_basic_services: true` . This turns a legal or ethical commitment into a computational invariant that can be mechanically verified. This approach is consistent with proposals to embed human rights into digital technologies across their lifecycle, using them to guide development and deployment [54] . Finally, to protect the integrity of the system's core principles, contamination-resistant manifests are necessary. The `aln-rights-validator` tool, mentioned in the provided learnings, is the perfect instrument for this purpose . This tool would act as a pre-commit hook or a mandatory CI job to prevent any unauthorized modification or removal of critical neurorights clauses, such as `noexclusion_basic_services` or `noscore_from_inner_state`, by automated rewriting tools or malicious actors . This ensures that the most fundamental guarantees remain untouchable and preserved throughout the software's evolution [56] .

The following table summarizes the key technical and governance initiatives, highlighting their interconnectedness.

Initiative Area	Specific Task	Technology / Method	Governance Impact
Code Generation	Develop procedural macros (`alnbind!`) for full ALN-to-Rust translation.	`proc-macro2`, `quote`, `syn` [60] [62]	Creates a deterministic, repeatable, and auditable link between policy and implementation.
Guard Coverage	Audit and wrap all sensitive execution paths to flow through generated guards.	Static analysis, risk-based assessment [53]	Ensures all actions are vetted against safety and rights policies, closing attack vectors.
Monotone OTA Tooling	Implement property-based tests and explore formal verification (Creusot/Kani).	`proptest`, Creusot, Verus [37] [49]	Provides strong, evidence-backed guarantees that system updates maintain safety and fairness.
Hex-Traced Logs	Standardize and implement `hex_trace` in all `DecisionGuardOutcome` returns.	Deterministic hashing, cryptographic signing	Enables transparent, auditable decision-making and supports the right to contest AI decisions.
Proofs of Non-Exclusion	Formally prove correctness of `EqualityPaymentGuard` for basic services.	Formal methods, model checking [51]	Transforms the promise of non-exclusion into a computational invariant, enforceable by the system.
Contamination-Resistant Manifests	Implement `aln-rights-validator` as a mandatory pre-commit/C.I. check.	Custom CLI tool, schema validation [67]	Protects fundamental neurorights from being altered or removed, preserving the integrity of the system's core values.

By pursuing these technical and governance initiatives in parallel, the project can build a system that is not only highly functional and safe but also deeply trustworthy. The emphasis on "audit-readiness by construction" ensures that transparency and accountability are woven into the fabric of the technology from the very beginning, fulfilling the core research goal.

Synthesis and Actionable Roadmap

The comprehensive analysis of the ALN-to-Rust workflow reveals a clear, coherent, and technically sound strategy for building a bioscale-compliant, safety-first, and rights-aware system. The overarching goal is to transition from a reactive model of safety—where checks are applied at runtime—to a proactive, preventative model where unsafe states are made unrepresentable at the type level. This is achieved through a three-layered architectural approach: a foundational layer of hardened safety enforced via ALN-as-type-system principles; an architectural innovation of canonical, referenceable neurorights envelopes; and a final layer of observability and governance through metrics integration and CI/manifest validation. This synthesis culminates in an actionable roadmap designed to systematically close technical gaps while embedding auditability and accountability into every stage of the development lifecycle.

The foundational principle of treating ALN as a formal type system is the most critical element of this strategy. By leveraging procedural macros and a pre-compile ALN validator, the system can generate Rust code that omits functions for prohibited actions (e.g., stimulation when `stimulation_allowed` is false) and enforces monotonic safety inequalities (`G_new <= G_old`) at compile time [62]. This approach, inspired by safety-critical embedded Rust projects, shifts the burden of proof from runtime assertion to compile-time guarantee, drastically reducing the potential for human error and latent vulnerabilities [47] [51]. The success of this foundational layer is paramount, as all subsequent features depend on its integrity.

Architecturally, the decision to anchor transhuman rights fields in a canonical `neurorights.envelope.citizen.v1` shard is a masterstroke of design. This pattern of referencing a shared invariant layer instead of duplicating data across policy stanzas ensures consistency, simplifies policy definitions, and creates a single, verifiable target for formal proofs of non-exclusion and neuro-safety [54]. The `TranshumanRightsEnvelope` struct in Rust provides the concrete implementation of this concept, serving as the common data model for all policy enforcement guards . This directly translates abstract legal and ethical principles into computable, machine-checkable contracts.

Finally, the layered workflow of metrics integration followed by mandatory CI/manifest validation completes the picture. The integration of bounded eco-civic metrics like `EcoImpactScore` enriches the system with a dimension of social accountability without compromising the hardened safety envelope . The CI pipeline, acting as an immutable gatekeeper, operationalizes the enforcement of the ALN contract by validating every change against the rules of the validator and the manifest linter . This creates a closed-loop system where every artifact—from the initial ALN shard to the final deployed binary—is traceable, verifiable, and compliant with the established safety and rights policies, embodying the principle of "audit-readiness by construction" [53] .

Based on this synthesis, the following actionable roadmap is recommended to guide the refinement of the ALN-to-Rust workflow:

1. **Prioritize Code Generation Engine Development:** The immediate focus must be on completing the procedural macro-based code generator. This involves implementing the `alnbind!` and `metricderive!` macros using the `syn` and `quote` crates to handle the full spectrum of ALN-to-Rust translation. This engine is the central nervous system of the entire workflow.

2. **Build the Static ALN Validator:** Concurrently, develop the static ALN validator tool. This tool must perform two primary functions: rejecting schemas with forbidden actuation fields (`torque`, `stimpattern`) and verifying that all monotonic safety inequalities are upheld. This tool will serve as the first line of defense against invalid configurations.

3. **Implement the Core Neurorights Crate:** Create the `crates/transhuman-rights-core` with the `TranshumanRightsEnvelope` struct and a default `NeurorightsGuard` implementation. This crate will serve as the canonical reference for rights enforcement across the system. Begin by defining the `neurorights.envelope.citizen.v1` ALN particle.

4. **Refactor Policies and Migrate Guard Coverage:** Initiate a systematic refactoring of existing ALN policy stanzas to replace duplicated rights fields with references to the new canonical envelope shard. Following this, conduct a comprehensive audit of the Rust codebase to ensure all sensitive execution paths are funneled through the generated guards, wrapping any remaining unguarded logic.

5. **Design and Implement the CI Pipeline:** Define the exact schema for the research manifest and configure the CI pipeline (e.g., GitHub Actions) to execute the regeneration, linting, and testing workflow. Implement the manifest linter as a mandatory gating step to prevent non-compliant changes from being merged.

6. **Develop Governance Artifacts:** Co-develop the hex-traced decision logger and the `aln-rights-validator` tool. Standardize the `hex_trace` field to provide auditable proofs of decisions. Use the validator to protect the integrity of the core neurorights clauses.

7. **Investigate Advanced Verification Techniques:** As a long-term goal, investigate the integration of formal verification tools like Kani or Prusti to supplement property-based tests. Focus initial efforts on proving the correctness of the most critical safety and fairness invariants, such as those governing basic service access and OTA updates.

By following this roadmap, the project can systematically build a robust, auditable, and ethically-grounded system. The proposed ALN-to-Rust workflow is not merely a technical exercise; it is a blueprint for engineering a future where technology is fundamentally aligned with human safety, dignity, and rights.

Reference

1. time - How to benchmark programs in Rust? https://stackoverflow.com/questions/13322479/how-to-benchmark-programs-in-rust
2. Arxiv今日论文| 2025-12-25 - 闲记算法 http://lonepatient.top/2025/12/25/arxiv_papers_2025-12-25
3. 机器学习2025_2_5 https://www.arxivdaily.com/thread/63859
4. Computer Science May 2025 https://www.arxiv.org/list/cs/2025-05?skip=4150&show=2000
5. A Framework-Driven Evaluation and Survey of MCU Fault ... https://www.researchgate.net/publication/397536500_A_Framework-Driven_Evaluation_and_Survey_of_MCU_Fault_Injection_Resilience_for_IoT
6. ALTA 2019 Proceedings of the 17th Workshop of the ... https://aclanthology.org/U19-1.pdf
7. Automated Proof Generation for Rust Code via Self-Evolution https://arxiv.org/html/2410.15756v2
8. A Tool-Assisted Approach to Engineer Domain-Specific ... https://dl.acm.org/doi/pdf/10.1145/3550356.3563133
9. (PDF) A tool-assisted approach to engineer domain- ... https://www.researchgate.net/publication/365269527_A_tool-assisted_approach_to_engineer_domain-specific_languages_DSLs_using_Rust
10. TYPE-CONSTRAINED CODE GENERATION WITH ... https://openreview.net/pdf/fcec8d95bee1e22da4297bbe39c40960fe62ec27.pdf
11. Multi-Agent Code Generation and Problem Solving through https://aclanthology.org/2025.findings-naacl.285.pdf
12. (PDF) Free will, neurosciences & robotics https://www.researchgate.net/publication/391865763_Free_will_neurosciences_robotics
13. Artificial Intelligence and Speech Technology https://link.springer.com/content/pdf/10.1007/978-3-031-75167-7.pdf
14. (PDF) Citizen-centred approach to public engagement on ... https://www.researchgate.net/publication/398485483_Citizen-centred_approach_to_public_engagement_on_the_ELSI_of_health_technologies
15. 机器学习2025_1_28 https://www.arxivdaily.com/thread/63555

16. Machine translation of cortical activity to text with an encoder ... https://pmc.ncbi.nlm.nih.gov/articles/PMC10560395/
17. (PDF) Security Challenges in Brain-Computer Interfaces ... https://www.researchgate.net/publication/394535009_Security_Challenges_in_Brain-Computer_Interfaces_and_Neuro-Wearables
18. Orthogonal Representations of Object Shape and Category ... https://pmc.ncbi.nlm.nih.gov/articles/PMC7016009/
19. The Evolution of Legal Personhood and Its Implications for ... https://www.researchgate.net/publication/395346229_Beyond_Personhood_The_Evolution_of_Legal_Personhood_and_Its_Implications_for_AI_Recognition
20. PeerSync: Accelerating Containerized Model Inference at ... https://arxiv.org/html/2507.20116v2
21. A portable diagnosis and distance localization approach ... https://www.sciencedirect.com/science/article/abs/pii/S1474034623003749
22. Lucene Change Log https://lucene.apache.org/core/9_12_2/changes/Changes.html
23. Semantic Communication Unlearning: A Variational ... https://www.mdpi.com/1999-5903/18/1/17
24. network-and-edge-reference-system-architectures-portfolio ... https://builders.intel.com/docs/networkbuilders/network-and-edge-reference-system-architectures-portfolio-user-manual-1707468897.pdf
25. Novel Neurorights: From Nonsense to Substance - PMC https://pmc.ncbi.nlm.nih.gov/articles/PMC8821782/
26. 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
27. Rust设计模式：sealed trait-腾讯云开发者社区 https://cloud.tencent.com/developer/article/2413467
28. Differential Symbolic Testing for LLM-Transpiled C-to-Rust ... https://arxiv.org/pdf/2510.07604
29. GenC2Rust: Towards Generating Generic Rust Code from C https://dl.acm.org/doi/pdf/10.1109/ICSE55347.2025.00127
30. TYPE-CONSTRAINED CODE GENERATION WITH ... https://openreview.net/pdf?id=LYVyioTwvF
31. masters theses in the pure and applied sciences https://link.springer.com/content/pdf/10.1007/978-1-4615-2832-6.pdf

32. (PDF) Environmental Impact Assessement methodology [Y. ... https://www.academia.edu/48936044/Environmental_Impact_Assessement_methodology_Y_Anjaneyulu_Valli_Manickam_
33. 333333 23135851162 the 13151942776 of 12997637966 ftp://ftp.cs.princeton.edu/pub/cs226/autocomplete/words-333333.txt
34. Compare Packages Between Distributions https://distrowatch.com/dwres.php?resource=compare-packages&firstlist=gentoo&secondlist=devuan&firstversions=0&secondversions=0&showall=yes
35. Count 1w100k | PDF | Internet Forum https://www.scribd.com/document/749397253/count-1w100k
36. Type-Constrained Code Generation with Language Models https://arxiv.org/pdf/2504.09246?
37. Creusot: a Foundry for the Deductive Verification of Rust ... https://inria.hal.science/hal-03737878v1/document
38. TYPE-CONSTRAINED CODE GENERATION WITH ... https://openreview.net/pdf/fcec8d95bee1e22da4297bbe39c40960fe62ec27.pdf?utm_source=chatgpt.com
39. (PDF) KRust: A Formal Executable Semantics of Rust https://www.researchgate.net/publication/324859562_KRust_A_Formal_Executable_Semantics_of_Rust
40. Using const defined in generic trait-bound types in Rust https://stackoverflow.com/questions/78521572/using-const-defined-in-generic-trait-bound-types-in-rust
41. Source Packages in "sid", Subsection misc https://packages.debian.org/source/sid/misc/
42. Paquets source dans « sid », Sous-section misc https://packages.debian.org/fr/source/sid/misc/
43. Experts for Nodejs-Mobile-React-Native Plugins Readme https://www.linknovate.com/search/?query=nodejs-mobile-react-native%20plugins%20readme
44. NixOS https://distrowatch.com/table-mobile.php?distribution=nixos&pkglist=true&version=unstable
45. Google Books Common Words | PDF | Experience https://www.scribd.com/document/852044092/Google-Books-Common-Words
46. Leveraging rust types for modular specification and ... https://dl.acm.org/doi/10.1145/3360573
47. Environment for the Deductive Verification of Rust Programs https://inria.hal.science/hal-03526634/document

48. Automated Proof Generation for Rust Code via Self-Evolution https://www.researchgate.net/publication/385107421_Automated_Proof_Generation_for_Rust_Code_via_Self-Evolution
49. automated proof generation for rust code https://arxiv.org/pdf/2410.15756?
50. rust - How to use a macro to generate compile-time unique ... https://stackoverflow.com/questions/71463576/how-to-use-a-macro-to-generate-compile-time-unique-integers
51. Generating Automotive Code: Large Language Models ... https://arxiv.org/html/2506.04038v1
52. A Case Study on Integrating an Eco-Design Tool into the ... https://www.mdpi.com/2076-3417/14/22/10583
53. Software Development Process for Safety-Critical Systems https://visuresolutions.com/alm-guide/software-development-process-for-safety-critical-systems/
54. Shaping a rights-oriented digital transformation (EN) https://www.oecd.org/content/dam/oecd/en/publications/reports/2024/06/shaping-a-rights-oriented-digital-transformation_30378a18/86ee84e2-en.pdf
55. (PDF) Versioned linking of semantic enrichment of legal ... https://www.researchgate.net/publication/263102337_Versioned_linking_of_semantic_enrichment_of_legal_documents_Emerald_An_implementation_of_knowledge-based_services_in_a_semantic_web_approach
56. GDPR compliance via software evolution: Weaving security ... https://www.sciencedirect.com/science/article/pii/S0164121224001894
57. Features of Regulation Document Translation into a ... https://www.mdpi.com/2673-4109/4/2/22
58. Zero-Shot Cross-Lingual Transfer in Legal Domain Using ... https://arxiv.org/pdf/2111.14192
59. Using model-driven engineering to automate software ... https://link.springer.com/article/10.1007/s10515-024-00419-y
60. Procedural Macros - The Rust Reference https://rustwiki.org/en/reference/procedural-macros.html
61. Lucene Change Log https://lucene.apache.org/core/10_2_1/changes/Changes.html
62. Rust 过程宏（Procedural Macros）实战：syn 与quote 的元 ... https://blog.csdn.net/weixin_75201252/article/details/154154854
63. How do I make the quote! macro print hex literals? https://stackoverflow.com/questions/78902816/how-do-i-make-the-quote-macro-print-hex-literals

64. How to import all macros, derives, and procedural ... https://stackoverflow.com/questions/50999749/how-to-import-all-macros-derives-and-procedural-macros-in-rust-2018-without-us
65. Const Generics https://practice.course.rs/generics-traits/const-generics.html
66. Replace a model in openapi schema with external Rust ... https://stackoverflow.com/questions/78564775/replace-a-model-in-openapi-schema-with-external-rust-value-during-generation
67. 平台和工具 - 架构师研究会 https://architect.pub/book/export/html/36
68. hw3_stats_google_1gram.txt https://www.cs.cmu.edu/~roni/11661/2017_fall_assignments/hw3_stats_google_1gram.txt
69. 333333 23135851162 the 13151942776 of 12997637966 https://www.cs.princeton.edu/courses/archive/spring25/cos226/assignments/autocomplete/files/words-333333.txt
70. Using $crate in Rust's procedural macros? https://stackoverflow.com/questions/44950574/using-crate-in-rusts-procedural-macros
71. 2015 Regulation | PDF | University And College Admission https://www.scribd.com/document/380728873/2015-Regulation