Sovereign Evolution: A Dual-Layer Rust/ALN Framework for Verifying Neuron-Particle Workflows Against Eibon Governance

This report presents a comprehensive research analysis and architectural blueprint for developing a secure, sovereign-compatible Rust/ALN framework designed for 'neuron-particle' branches within cybernetic-blockchain ecosystems. The central objective is to establish a robust mechanism that ensures continuous operation of critical workflows and neuron-based services, all while enforcing strict Eibon governance through a dual-layer model of compile-time invariant enforcement and dynamic runtime re-verification. The framework is engineered to be compatible with a diverse set of execution environments, prioritizing self-hosted, ALN-governed runners over GitHub-hosted ones, thereby anchoring trust in a sovereign registry and Cybercore-Brain graph rather than public infrastructure. This research addresses the dual nature of its intended use cases: serving as the foundation for CI/CD pipelines that validate, build, and publish artifacts via a six-stage 'donut-loop', and as the core enforcement layer for persistent runtime services such as BCI routers and nanoswarm backends. These services must consume published artifacts and dynamically enforce bioscale safety invariants against live telemetry. The solution hinges on a tightly integrated system of typed Rust structures mirroring Assurance Logic Network (ALN) particles, a suite of custom procedural macros for static checking, formal verification with tools like Kani, and a runtime attestation model that validates compliance before every potentially sensitive operation. Key entities include the `NeuronParticleBranch` representing an evolutionary unit, the `QuantumConsciousnessEnvelope` for speculative quantum-inspired safety corridors, and the `ALNComplianceParticle` which acts as the primary gatekeeper for all host-facing actions. All upgrades are anchored to a scientifically-grounded `EvidenceBundle` to ensure adherence to established biophysical principles.

Architectural Foundations: Sovereign Control Through Deconstructed Trust

The foundational principle underpinning the proposed framework is the deliberate deconstruction of trust in external infrastructure, most notably GitHub, and its replacement with a self-contained, auditable, and sovereign governance model. The architecture is not merely a technical specification but a strategic response to the inherent vulnerabilities of relying on centralized cloud services for the development and deployment of advanced cybernetic systems [6] [31]. The core strategy involves shifting the locus of trust from the platform that executes jobs (GitHub) to the artifacts that define the rules of engagement (ALN particles), the environment where those jobs run (governed runners), and the global graph that verifies their relationships (Cybercore-Brain). This approach ensures that even if GitHub were to become unavailable or compromised, the core logic and control of the evolution process would remain intact and functional within the sovereign cluster .

The project reframes GitHub's role from a source of truth to a simple orchestrator. It is treated as a "hosted runner," a client, or a mirror service whose primary function is to trigger workflows based on events like pull requests or scheduled cron jobs [2] . The actual validation, building, testing, and signing of artifacts occur exclusively on a fleet of self-hosted runners operating under the direct control of the sovereign entity . These runners are located within a trusted infrastructure cluster, such as VSC-ARTEMIS or the Phoenix lab, and are not implicitly trusted simply because they are associated with a GitHub repository . This separation of orchestration from execution is a critical design choice for achieving sovereignty. The trust root is explicitly defined as the combination of the sovereign ALN-first registry, the Cybercore-Brain graph, and the Phoenix jurisdiction particle, not GitHub itself . This ensures that the validity of any workflow or neuron-particle is determined by its adherence to this internal, immutable governance framework, regardless of its origin .

This sovereign model is reinforced by a rigorous two-tier enforcement mechanism that operates at both the creation phase (CI/CD) and the execution phase (runtime). The first tier, compile-time and CI/CD invariant enforcement, builds a "safe-by-construction" foundation by using ALN schema checks, custom Rust proc-macros, and formal verification with Kani to mathematically prove properties about the code before it can ever be released . Any artifact that fails to meet these stringent criteria is rejected by the CI pipeline. The second tier, dynamic runtime re-verification, ensures that deployed systems remain compliant with evolving governance policies in real-time. Before any host-facing operation—such as a BCI pulse, XR loop, or nanoswarm actuation—a runtime safety router inspects an `ALNComplianceParticle`. This particle is a cryptographic hash of the current state, including the upgrade descriptor, consent flags, budget compliance, and evidence bundle . The router dynamically re-verifies this particle against live host telemetry (e.g., EEG load, HRV, temperature) and the active set of governance particles fetched from the sovereign registry . This defense-in-depth strategy combines the certainty of formal methods with the adaptability of runtime attestation, creating a robust shield against both static flaws introduced during development and dynamic threats that may emerge during operation [34] [37] .

The enforcement of this sovereign model extends to the very identity and environment of the runners themselves. Instead of assuming a generic Linux environment, the framework introduces ALN artifacts like `SessionToken` for authentication and `EnvGrant` for authorization . An `EnvGrant` is a typed particle that defines the precise environment in which a job is allowed to run, including fields for CPU architecture, operating system, maximum memory in megabytes, and a list of allowed crates . When a job is dispatched to a sovereign runner, regardless of whether the original request came from a self-hosted machine or a GitHub-hosted runner, the runner first validates the incoming `EnvGrant` against its own capabilities and policies . If the grant is incompatible—for instance, if it requests a crate not available on the runner or a memory limit exceeding the hardware's capacity—the job is rejected . This creates a powerful boundary, preventing any job, even one triggered by a compromised account on a public runner, from running outside its explicitly permitted sandbox. This aligns with principles of trustless autonomy, where nodes coordinate through distributed protocols and cryptographic proofs rather than a central authority [4] . By coupling local protection enforcement with remote attestation, the system can verify that a device is behaving as intended without needing to implicitly trust the platform it runs on [11] [24] .

To manage this complex interplay between different parts of the system, the architecture mandates the creation of a "spec-sync" tool . This tool acts as a consistency checker, failing the CI pipeline if there is any drift between the various components. For example, it would fail if an ALN fragment defining a clause ID is changed without corresponding updates to the Rust guard modules, Prometheus metric names, or the manifest schema . This automated validation prevents subtle but critical inconsistencies from creeping into the system, which could otherwise lead to security holes or governance bypasses. The entire CI process is orchestrated around a standardized six-stage 'donut-loop' (Ingest, Validate, Transform, Index, Release, Monitor), where each stage is annotated with a `ci.workline.zerotrust.v1` particle to ensure consistent application of the governance rules across all repositories . Repositories emit their own `particlesexport.manifest.json` files, which are then ingested by the central Cybercore-Brain . Cybercore-Brain rebuilds the global graph from these manifests to assert that there are no dangling references and that every path capable of reaching a host is properly linked to Eibon neurorights clauses like `nononconsensualmodulation` or `rollbackanytime` . This creates a globally consistent view of the system's state and enforces a policy of zero-trust across the entire knowledge ring. The ultimate authority for high-impact decisions is further refined through a Blood/CHAT consensus model, where only Blood-activated roles (like Mentor/Teacher) can author or co-sign critical governance particles, and stakeholder scores influence the weight of artifacts in active workflows . This socio-technical layer ensures that the technical gates are aligned with a broader, community-vetted understanding of what constitutes safe and ethical evolution.

The Neuron-Particle ABI: Defining the Unit of Bioscale Evolution

At the heart of the proposed framework lies the concept of the "neuron-particle branch," which serves as the atomic unit of evolutionary change. This abstraction encapsulates a single, bioscale-safe upgrade or modification, complete with its associated invariants, metrics, and governance artifacts. To make this concept tangible and enforceable, the architecture defines a clear Application Binary Interface (ABI) that tightly couples Assurance Logic Network (ALN) particle schemas with mirrored Rust types and procedural macros. This ABI is not just a translation layer; it is the primary mechanism for propagating governance rules from the abstract, human-readable ALN fragments down to the concrete, executable Rust code. The design ensures that any change to a governance rule expressed in ALN directly impacts the generated Rust guards, creating a strong, verifiable link between policy and implementation.

The foundation of the ABI is the definition of new ALN particles that model the key entities of the neuron-particle system. A central particle is `neuron.particle.branch.v1`, which represents a complete evolutionary window . Its fields are designed to capture all essential metadata for a given day's cycle of upgrades. These fields include the `host_did` (Decentralized Identifier of the host), the `bostrom_addr` (an address on the Bostrom network), a list of `upgrade_ids`, resource consumption metrics like `energy_j` (in Joules) and `protein_aa` (in amino acids), and behavioral constraints such as the `thermo_corridor` (a thermodynamic envelope) and `ml_schedule` (a machine learning duty schedule) . Crucially, this particle also contains the `git_ref` of the commit being built, the `runner_id` where it was executed, and the `workflow_id` that orchestrated its creation . This comprehensive record transforms the neuron.particle.branch.v1 into a cryptographically verifiable claim about a specific piece of code, its resources, and its context of production. Another key particle is the quantum.consciousness.envelope.v1, which introduces a speculative but structurally important concept . Its fields, such as cog_load_index, stability_margin, and entropy_budget, are intended to represent safety corridors for consciousness-like processes, tying them to the BrainSpecs and the underlying biophysical evidence registry . These ALN particles are not just data structures; they are legal and physiological surfaces that the system's macros will compile into actual Rust guards .

These ALN particles are mirrored precisely in a dedicated Rust crate, crates/neuron-particle-abi/src/lib.rs . This crate defines Rust structs that correspond directly to the ALN schemas. For example, NeuronParticleBranch mirrors neuron.particle.branch.v1, and QuantumConsciousnessEnvelope mirrors its counterpart . This mirroring is bidirectional. The bioscale-evolution-cli tool can generate Rust code from ALN fragments, and conversely, Rust code can be analyzed to generate or update ALN schemas. This tight coupling is essential for maintaining consistency. Furthermore, the ABI reuses existing types from other crates, such as UpgradeDescriptor, HostBudget, ThermodynamicEnvelope, MlPassSchedule, and ReversalConditions from crates/bioscale-upgrade-store . This promotes modularity and avoids reinventing the wheel, leveraging a shared foundation for representing core concepts like budgets and envelopes across different subsystems. The EvidenceBundle type is also part of this ABI, designed to hold the 10-tag evidence chain that anchors every upgrade to specific biophysical domains .

The true power of the ABI is realized through a suite of procedural macros defined in the crates/bioscale-upgrade-macros crate . These macros provide a high-level, domain-specific language (DSL) for developers to declare neuron-particle upgrades in a way that is both human-readable and machine-enforceable. The primary macro, #[bioscale_upgrade(id = "...", evidence = "...", neuron_branch = "...")], is an attribute macro that can be applied to Rust structs or enums representing an upgrade . When the compiler processes this macro, it automatically implements the Into<UpgradeDescriptor> and Into<NeuronParticleBranch> traits for the annotated item . This means that any developer-defined upgrade type can be seamlessly converted into the canonical forms used by the rest of the system. The macro also performs several crucial compile-time actions. It pulls default values from a registry, injects a full EvidenceBundle containing all ten required evidence tags, and ensures that the specified neuron_branch ID is correctly referenced . This automates the tedious and error-prone task of manually constructing these complex objects and guarantees that every upgrade adheres to the basic structural requirements.

Beyond high-level declaration, the framework provides field-level attributes and constraint macros to enforce granular invariants. For instance, a developer might use `#[quantum_cog_load(max = 0.35)]` on a field within an upgrade struct . This attribute lowers into the corresponding field of the `QuantumConsciousnessEnvelope` that will be generated for that upgrade . Similarly, constraint macros like `aln_enforce_duty!(duty = 0.4)` and `aln_enforce_qcon!(stability_margin = 0.3)` are used to constrain const generics on types like `StimChannel` or neuron-kernel functions . If a developer attempts to instantiate a `StimChannel` with a duty cycle higher than the one enforced by the macro, the code will fail to compile . This pushes the enforcement of physical and logical limits from runtime checks into the compiler itself, catching errors at the earliest possible stage. The framework also introduces a new trait, `NeuronParticleKernel`, in a guard crate like `crates/neuron-particle-guard/src/lib.rs` . This trait might define methods like `fn admissible(&self, env: &QuantumConsciousnessEnvelope, host: &HostBudget) -> bool`, providing a standardized interface for evaluating whether a given kernel (the core computational logic of an upgrade) is admissible under the current environmental and conscious-state constraints . This trait-based approach allows for polymorphic behavior, enabling the runtime safety router to evaluate different types of neuron-particle kernels through a common interface, mirroring patterns already used for BCI, nanoswarm, and neuromorphic corridor enforcement .

The following table summarizes the key components of the neuron-particle ABI, illustrating the tight relationship between the ALN particle schemas and their Rust counterparts.

ALN Particle Schema	Rust Crate & Type	Purpose and Fields
neuron.particle.branch.v1	crates/neuron-particle-abi::NeuronParticleBranch	Represents a single daily evolutionary unit. Fields: host_did, bostrom_addr, upgrade_ids[], energy_j, protein_aa, thermo_corridor, ml_schedule, evidence_hex[], git_ref, runner_id, workflow_id.
quantum.consciousness.envelope.v1	crates/neuron-particle-abi::QuantumConsciousnessEnvelope	Defines safety corridors for consciousness-related computations. Fields: cog_load_index, stability_margin, entropy_budget, rollback_threshold.
ci.github.runner.profile.v1	crates/ci-neuron-bridge::RunnerProfile	Describes the capabilities and restrictions of a CI runner. Fields: runner_id, os, arch, max_mem_mb, max_parallel_jobs, allowed_crates[], evidence_hex[].
ci.github.workflow.evolution.v1	crates/ci-neuron-bridge::WorkflowParticle	Links a GitHub workflow to a neuron-particle evolution unit. Fields: workflow_id, repo, branch, neuron_branch_ids[], aln_clause_ids[], kani_profiles[].
ALNComplianceParticle	crates/bioscale-upgrade-store::ALNComplianceParticle	A cryptographic hash acting as a runtime gate for host-facing operations. Fields: Hash of UpgradeDescriptor, BrainSpecs, BciHostSnapshot, DID, neurorights flags, evidence tags.

This structured ABI ensures that every aspect of a neuron-particle's lifecycle—from its initial definition in ALN, through its generation of Rust code, its compilation and verification, to its final execution—is governed by a consistent and verifiable set of rules. It transforms governance from a document into a compile-time and runtime enforceable property of the software itself.

Compile-Time Enforcement: A Two-Tier Invariant Model with Proc-Macros and Kani

The framework's first line of defense is a sophisticated, multi-layered compile-time and CI/CD invariant enforcement model. This tier is responsible for constructing a "safe-by-construction" foundation, ensuring that non-compliant, unsafe, or logically flawed code is prevented from ever entering the release pipeline. It achieves this through a synergistic combination of ALN schema anchoring, a rich set of custom Rust procedural macros, and formal verification using the Kani theorem prover. This model leverages Rust's strengths in memory safety and compile-time computation to embed governance directly into the language [18], treating the compiler as an active participant in the security and correctness process.

The enforcement begins with mandatory ALN schema checks performed by a CI job named `aln-spec-check` . Every `.aln` file fragment that describes a component of the system—including neuron-particles, CI profiles, or governance clauses—must be explicitly "cyberlinked" to a set of baseline governance particles . These anchor particles include `greatperplexity.core.v1`, `ci.workline.zerotrust.v1`, and a specific jurisdiction particle like `policy.jurisdiction.us-az-maricopa-phoenix.v1` . This creates an unbreakable chain of trust, rooting every new piece of governance logic back to a sovereign, audited authority. If a new ALN fragment fails to link to these required particles, the CI job immediately fails, preventing the introduction of orphaned or unauthorized governance rules. This step ensures that the entire system's policy landscape remains coherent and controlled.

Building upon this schema foundation, the framework employs a suite of procedural macros defined in the `crates/bioscale-upgrade-macros` crate to provide a high-level DSL for developers . These macros translate declarative intent into concrete, verifiable Rust code and logic. The `#[bioscale_upgrade]` attribute macro is the cornerstone of this system. When placed on a Rust struct, it generates implementations for traits like `Into<UpgradeDescriptor>`, automating the construction of the canonical upgrade representation . More importantly, it performs compile-time validation. For example, it can be configured to enforce that every upgrade carrying a certain evidence tag must also specify a `ReversalConditions` struct, failing compilation if this contract is violated . Similarly, field-level attributes like `#[bioscale_energy_mul(0.03)]` and `#[bioscale_protein_mul(1500000)]` are not just metadata; they lower into fields within the generated `HostBudget` and `ReversalConditions` structs, ensuring that cost and rollback calculations are explicitly defined and tied to the upgrade . Constraint macros like `aln_enforce_duty!(duty = 0.4)` and `aln_enforce_qcon!(stability_margin = 0.3)` are used to constrain const generics on types like `StimChannel` or neuron-kernel functions, turning physical limitations into compile-time errors . This moves the enforcement of critical invariants from runtime checks, which can be bypassed or fail silently, into the compiler, which offers absolute guarantees.

For critical upgrades deemed high-risk, the framework escalates from static analysis to formal verification using Kani, a C and Rust verifier . The CI pipeline includes dedicated jobs that run Kani harnesses against the code for these upgrades. A Kani harness is a small test program that defines a finite state machine representing the interactions between key safety-critical structs: `HostBudget`, `BrainSpecs`, `BciHostSnapshot`, and the ten key safety structs like `CognitiveLoadEnvelope` and `BioKarmaRiskVector` . The harness proves specific, user-specified properties about this state machine. For example, it can prove that "no sequence of states can result in an energy or protein budget violation" or that "any time a reversal condition threshold is breached, an `EvolutionAuditRecord` is emitted" . Kani exhaustively explores the state space of the harness and can either find a counterexample (a bug) or provide a mathematical proof that the property holds for all possible inputs and execution paths. This level of assurance goes far beyond traditional unit or property-based testing, providing a near-certainty of correctness for the most critical safety properties. The CI pipeline is configured to require at least one valid Kani harness for every critical upgrade, ensuring that high-stakes changes receive the highest level of scrutiny .

The output of this entire compile-time and CI process is a set of machine-readable artifacts that serve as a canonical record of the evolution window. The `bioscale-evolution-cli` tool is responsible for generating a `research/YYYY-MM-DD-manifest.json` file for each daily cycle . This manifest is a comprehensive JSON object that captures the state of the world at the time of the build. Its schema is validated rigorously in CI, with checks that include:

- **Evidence Integrity:** Ensuring that every entry in the \`upgrades\` array has exactly 10 evidence hex tags, all of which are drawn from a pre-approved registry (e.g., \`a1f3c9b2\` through \`8f09d5ee\`) or a versioned extension thereof .
- **Envelope Validation:** Confirming that numerical values within thermodynamic, ML, and quantum consciousness envelopes fall within globally defined safe ranges, derived from sources like \`DefaultBioPhysEvidence\` and quantum-geometry safety bounds .
- **Contract Verification:** Asserting the presence of at least one corresponding unit test for each upgrade and, for critical upgrades, the existence of a successful Kani harness job in the CI logs .
- **Metadata Capture:** Including essential metadata such as the \`host_did\`, Bostrom address, git commit hash, and versions of all relevant crates .

Once validated, this manifest is anchored to the blockchain via Googolswarm/DID transaction proofs, creating an immutable and publicly verifiable link between the evolution window and the neurorights and audit logs governed by Eibon . This entire two-tier compile-time enforcement model—combining ALN anchoring, macro-driven static checks, and Kani-powered formal proofs—creates a formidable barrier against introducing unsafe or non-compliant code. It ensures that the only artifacts that can be released are those that have been proven, to a high degree of certainty, to adhere to the governing bioscale and legal invariants.

Dynamic Runtime Re-Verification: Enforcing Compliance on Every Operation

While the compile-time and CI/CD enforcement model establishes a safe foundation, it cannot account for the dynamic nature of governance or the changing state of the host environment. Policies evolve, stakeholder scores are updated, and a host's physiological state fluctuates in real-time. To address this, the framework implements a second, equally critical tier of enforcement: dynamic runtime re-verification. This mechanism ensures that even after a neuron-particle has been successfully built and released, it must continuously prove its right to operate. Every host-facing action, from a BCI pulse to a nanoswarm command, is gated by a check that validates its compliance against the *live* set of governance particles and the *current* state of the host. This creates a defense-in-depth strategy where safety is not a one-time check at release but a continuous, ongoing process of attestation.

The primary instrument of runtime enforcement is the `ALNComplianceParticle`. This is a specialized ALN particle that acts as a "may-this-run?" ticket for any operation involving a neuron-particle . Before a safety router forwards a request to a backend (e.g., a BCI stimulator or a nanoswarm controller), it requires the request to be accompanied by a valid `ALNComplianceParticle` . This particle is a cryptographic hash that binds together a snapshot of the conditions under which the operation is requested. The fields hashed into this particle typically include: the hash of the `UpgradeDescriptor` being executed, the `BrainSpecs` of the host, the `BciHostSnapshot` (a real-time telemetry snapshot of the host's state), the host's Decentralized Identifier (DID), boolean flags indicating consent and neurorights status, a flag showing whether the `HostBudget` fits within the `ThermodynamicEnvelope`, and the `EvidenceBundle` that grounds the upgrade in science . Because this particle is a hash, it is computationally infeasible to forge a valid one for an unauthorized or non-compliant upgrade. The safety router simply needs to verify that the provided particle is valid according to the rules encoded in the `greatperplexity.core.v1` and other jurisdictional particles. This turns the problem of runtime safety from one of complex, stateful computation into one of efficient, cryptographic verification.

The runtime check does not end with the `ALNComplianceParticle`. The safety router performs a crucial additional step: it dynamically re-verifies the compliance of the operation against the live host state. For a BCI operation, this means taking the `BciHostSnapshot` included in the particle and comparing its telemetry against the thresholds defined in the `ThermodynamicEnvelope` and `ReversalConditions` structs that were used to generate the particle . For example, if the snapshot shows an elevated EEG load, low HRV, or excessive core temperature, the router's `snapshot_safe()` gate would reject the operation, even if the `ALNComplianceParticle` itself is perfectly valid . This ensures that the decision to allow an operation is not based solely on the state of the world when the upgrade was compiled, but on the state of the world at the moment of execution. This dynamic gating is extended to all types of implants and devices, including neuromorphic and organic systems, where live telemetry for duty, thermal load, metabolic rate, fatigue, and jitter is mapped to corresponding safety corridors defined in the ALN . This shared runtime clamp, where every actuation passes through the same Rust guard path, ensures a uniform and uncompromising standard of safety across all host-facing interfaces .

All decisions made by the runtime safety router are meticulously logged in `EvolutionAuditRecord`s and exposed as Prometheus metrics . This creates a transparent and auditable trail of every action attempted and every action blocked. Metrics are labeled with rich context, including `domain`, `policyid`, `upgradeid`, `hostid`, and even `runner_id` and `workflow_id` from the original CI build, allowing for end-to-end traceability . For instance, counters like `bcienvelope_breach_total{policyid,hostid}` and gauges like `qcon_stability_margin` provide real-time visibility into the health and safety of the system . Eibon monitors can query these metrics to verify that required workflows are firing on schedule and that safety gates are functioning as expected . If a critical workflow fails to execute within its allowed `max_gap_minutes` as defined in its `ci.github.workflow.evolution.v1` particle, the system can mark it as non-compliant, blocking further releases until the issue is resolved . This combination of immediate blocking, detailed auditing, and continuous monitoring provides a complete picture of the system's operational status.

The central nervous system of this runtime governance is the Cybercore-Brain, a global graph that continuously ingests manifests from all participating repositories . Periodically, Cybercore-Brain rebuilds the entire governance graph, asserting that every path that can reach a host is correctly wired to Eibon neurorights clauses like nononconsensualmodulation or noraweegexport . If it detects a miswired or stale governance path—perhaps a workflow that no longer has an associated neurorights template—it can flag it as a CI/runtime violation, forcing developers to correct the broken link . This global perspective ensures that local changes do not inadvertently introduce systemic risks. The authority of this entire system is ultimately backed by a Blood/CHAT consensus model, where only authorized roles can create or modify the foundational governance particles, and the influence of any artifact is weighted by stakeholder scores . This socio-technical layer ensures that the technical enforcement mechanisms are aligned with a community-vetted and evolving understanding of safety and ethics. Together, the ALNComplianceParticle, dynamic state checks, continuous auditing, and global graph validation form a robust runtime enforcement layer that keeps neuron-particles and their associated workflows actively and safely operational within the bounds of Eibon governance.

Infrastructure Strategy: Governing Self-Hosted Runners and Securing CI/CD Orchestration

The infrastructure strategy of this framework is a direct manifestation of its core principle of sovereignty. It deliberately prioritizes self-hosted, ALN-governed runners over GitHub-hosted runners, treating the latter as untrusted clients at best and optional mirrors at worst. This design choice is not merely an engineering preference but a fundamental security measure aimed at insulating the critical processes of building, validating, and releasing neuron-particle artifacts from the potential unreliability, compromise, or policy changes of public cloud platforms [6] [31]. The trust root is firmly planted within a sovereign-controlled infrastructure cluster, where every aspect of the execution environment is strictly regulated and verified.

The target infrastructure consists of a heterogeneous fleet of self-hosted runners, including bare-metal Linux servers, containers running on Kubernetes, and edge devices like NVIDIA Jetson systems . These runners are not generic; they are purpose-built and managed as part of the sovereign cluster. Their configuration is highly constrained to prevent privilege escalation or unintended side effects. A cornerstone of this strategy is the use of ALN artifacts for both authentication and authorization. Before a job can begin executing, it must present a SessionToken, which authenticates the identity of the agent requesting the job. Following authentication, the job must present an EnvGrant (environment grant), which authorizes what the job is allowed to do . The EnvGrant is a typed ALN particle that defines the precise sandbox in which the job will run. Its schema includes fields for the CPU architecture, operating system, maximum memory in megabytes, and a list of `allowed_crates` that the build process is permitted to access . This `CargoEnvDescriptor` is mirrored in Rust and used to validate the grant .

When a job is dispatched to a sovereign runner, the first action taken is to validate the incoming `EnvGrant` against the runner's own hard-coded policies and hardware capabilities . For example, a runner on a Jetson device would reject an `EnvGrant` that specifies an x86_64 architecture. A Kubernetes-based runner would check if the requested memory exceeds the pod's resource limits. A runner with a Blake3 quarantine spec would deny access to any crates listed in its Blake-family denylist [35]. This verification happens before any code is even checked out, effectively creating a hard boundary around the execution environment. This mechanism ensures that no matter where the job originated—from a private fork, a compromised account, or even a malicious pull request on a public repository—the execution is sandboxed within its approved confines [2]. This pattern of verifying the environment before execution is a classic zero-trust security practice, and its implementation here is tailored specifically to the unique constraints of cybernetic development.

GitHub Actions YAML files play a crucial role in this architecture, but their function is purely orchestration. They define the workflow steps and dependencies, but they do not control the execution environment . The YAML for a neuron-particle evolution job would look something like this:

```
jobs:
  build-and-test:
    runs-on: [self-hosted, linux]
    steps:
      - name: Build ABI and Guard Crates
        run: cargo build -p neuron-particle-abi -p neuron-particle-guard
      - name: Run Unit and Property Tests
        run: cargo test -p neuron-particle-guard
      - name: Run Kani Formal Verification
        uses: ./.github/actions/kani-runner
      - name: Emit and Validate Manifest
        run: ./scripts/generate_manifest.sh
```

The `runs-on: [self-hosted, linux]` directive signals that this job should be picked up by one of the controlled, sovereign runners. The actual commands are simple shell scripts that invoke the build tools. The heavy lifting of security and governance—of ensuring that the `kani-runner` action is legitimate, that `cargo` can only access approved crates, and that the job doesn't exceed its memory limit—is handled entirely by the sovereign runner and its `EnvGrant` validation logic, not by the GitHub Actions platform itself . This decouples the high-level workflow description from the low-level security enforcement, making the system resilient to changes in the CI platform.

The framework also anticipates scenarios where GitHub-hosted runners might still be used, for example, for less critical tasks like documentation generation or linter checks. In these cases, they are treated as untrusted clients . A job running on a GitHub-hosted runner would need to present a valid `SessionToken` and `EnvGrant` to interact with the sovereign infrastructure (e.g., to push a signed artifact to the ALN-first registry). If it fails to do so, it would be denied access. Alternatively, for truly read-only tasks, the runner might operate in a completely isolated mode, unable to affect any state in the sovereign cluster. This flexible yet strict approach allows for broad compatibility while never compromising the security of the core asset: the neuron-particle itself.

To manage the complexity of this multi-repo, multi-stage CI/CD pipeline, the architecture relies on a "spec-sync" tool . This tool is a critical piece of the automation infrastructure, designed to prevent the kind of drift that can break a system as complex as this one. It would run as a CI job before the main build and perform cross-cutting validations. For example, it would check that any change to an ALN fragment results in a corresponding update to the mirrored Rust types in the ABI crate. It would verify that every ALN clause ID used in a particle corresponds to a metric name in the Prometheus schema and appears in the manifest structure. It could even use APIs from proc-macro2 to introspect Rust code and cross-reference it with ALN schemas and manifest files [1] . By automating these consistency checks, the spec-sync tool reduces the cognitive load on developers and eliminates a major class of potential bugs related to misalignment between the different parts of the system. This focus on automated, holistic validation is essential for maintaining the integrity of the donut-loop and ensuring that every artifact flowing through the system is consistent, complete, and compliant.

Synthesis and Future Directions: From Speculation to Verified Implementation

This research report has detailed the design of a comprehensive, sovereign framework for the evolution of neuron-particle workflows. The core innovation lies in its systematic dismantling of trust assumptions regarding external infrastructure and its replacement with a self-contained governance model anchored in ALN particles, cryptographic verification, and a dual-layer enforcement strategy. The proposed architecture successfully addresses the user's primary goal by ensuring that critical workflows and services remain operational, protected, and fully compliant with Eibon governance. It achieves this through a tightly integrated system where compile-time checks, formal verification, and dynamic runtime attestation work in concert to create a defense-in-depth security posture.

The framework's strength is rooted in its two-tier enforcement model. The first tier, compile-time and CI/CD invariant enforcement, constructs a "safe-by-construction" foundation. By leveraging ALN schema anchoring, custom Rust procedural macros, and formal verification with Kani, it mathematically proves the absence of certain classes of bugs and guarantees adherence to bioscale and legal invariants before any code is released . This transforms the compiler and build system into active agents of security. The second tier, dynamic runtime re-verification, ensures that deployed systems remain compliant in a dynamic environment. Through the use of `ALNComplianceParticle` gates and continuous checks against live host telemetry, it prevents unsafe operations from ever reaching the hardware, adapting to real-time changes in both policy and physiology .

A key enabler of this entire system is the "neuron-particle," which serves as the atomic, auditable, and scientifically-grounded unit of evolution. The defined ABI, which maps ALN particle schemas to Rust types and macros, ensures a strong, verifiable link between abstract governance logic and concrete code . The requirement for a 10-tag `EvidenceBundle` per upgrade grounds the entire engineering process in established biophysical literature, moving it from arbitrary parameter tuning to evidence-based design . The strategic decision to prioritize self-hosted, ALN-governed runners further solidifies this sovereignty, insulating the critical build and validation processes from the potential volatility of public cloud platforms .

Despite the framework's completeness, several areas remain as opportunities for future research and clarification. First, the precise operational definition of "Eibon governance" warrants deeper exploration. While the term is used throughout, its interaction with the technical gates needs more detail. Clarifying whether it is a set of hardcoded rules, a dynamic smart contract system, or a consensus-driven process will be essential for implementing the runtime re-verification logic. Second, the `QuantumConsciousnessEnvelope` remains a highly theoretical construct [15]. Future work must focus on grounding its fields—such as `stability_margin` and `entropy_budget`—into concrete, measurable quantities that can be used for runtime checks. Finally, while the conceptual six-stage 'donut-loop' is well-defined, the practical engineering of the orchestrator that manages this complex, multi-repo workflow presents significant challenges [39]. Developing robust tooling, inspired by solutions in cloud-native computing like Kubernetes Operators [13] or frameworks for workload orchestration at the edge [36], will be necessary to bring this vision to life.

In conclusion, the proposed Rust/ALN framework provides a robust and compelling blueprint for creating a sovereign, secure, and auditable cybernetic development ecosystem. By integrating compile-time formal methods with dynamic runtime attestation and anchoring trust in a self-controlled registry and governance graph, it offers a viable path toward the safe and responsible evolution of advanced AI and cybernetic systems.

Reference

1. Rust proc macro to do compile time checking if two types ... https://stackoverflow.com/questions/78939065/rust-proc-macro-to-do-compile-time-checking-if-two-types-are-equivelant
2. How to protect github actions self-hosted runner? https://stackoverflow.com/questions/79107120/how-to-protect-github-actions-self-hosted-runner
3. Open Source Projects https://docs.daocloud.io/native/open/
4. Trustless Autonomy: Understanding Motivations, Benefits ... https://arxiv.org/html/2505.09757v2
5. The Annual AI Governance Report 2025 https://www.itu.int/epublications/en/publication/the-annual-ai-governance-report-2025-steering-the-future-of-ai/en/
6. Harnessing Artificial Intelligence in Social Security https://www.oecd.org/content/dam/oecd/en/publications/reports/2025/12/harnessing-artificial-intelligence-in-social-security_d03136ef/b52405c1-en.pdf
7. A Mathematical Theory of Communication: | Guide books https://dl.acm.org/doi/10.5555/1102016
8. Information Processing and Network Provisioning https://link.springer.com/content/pdf/10.1007/978-981-96-6462-7.pdf
9. (PDF) 3130 reviews WOS https://www.researchgate.net/publication/394486268_3130_reviews_WOS
10. Smart Indoor Farms: Leveraging Technological ... https://www.mdpi.com/2624-7402/3/4/47

11. Interoperable node integrity verification for confidential ... https://pdfs.semanticscholar.org/4bf6/7d52bba321f243f7ad7bdf6daabec9bb789b.pdf
12. TowardsDataScience-博客中文翻译-2019-二十九 https://www.cnblogs.com/apachecn/p/18462373
13. Leveraging APL and SPIR-V languages to write network ... https://inria.hal.science/hal-03155647/document
14. Computer Science https://arxiv.org/list/cs/new
15. Quantum Models of Consciousness from a ... https://arxiv.org/html/2501.03241v2
16. designing an integrated investment and risk governance ... https://www.researchgate.net/publication/398520514_DESIGNING_AN_INTEGRATED_INVESTMENT_AND_RISK_GOVERNANCE_FRAMEWORK_FOR_SOVEREIGN_PORTFOLIOS_A_PRACTICAL_MODEL_FOR_GOVERNMENT_INVESTMENT_ENTITIES
17. multi-page.txt https://documents1.worldbank.org/curated/en/663341468174869302/txt/multi-page.txt
18. Findings of the Association for Computational Linguistics https://aclanthology.org/volumes/2025.findings-emnlp/
19. article sustaining the environmental rights of children https://www.ohchr.org/sites/default/files/Documents/HRBodies/CRC/Discussions/2016/KarenMakuch.pdf
20. Corporate Income Taxes under Pressure - IMF eLibrary https://www.elibrary.imf.org/downloadpdf/display/book/9781513511771/9781513511771.pdf
21. Cryosphere and climate: the Arctic challenge https://unesdoc.unesco.org/ark:/48223/pf0000211941
22. Formal verification for high assurance security software in ... https://theses.hal.science/tel-05140376v2/file/Beurdouche_2020_These.pdf
23. 9.7 Release Notes | Red Hat Enterprise Linux | 9 https://docs.redhat.com/fr/documentation/red_hat_enterprise_linux/9/html-single/9.7_release_notes/index
24. Implementation of the TCG DICE Specification into the ... https://ieeexplore.ieee.org/iel8/6287639/6514899/11119633.pdf
25. What's new in Red Hat OpenShift https://www.redhat.com/en/whats-new-red-hat-openshift
26. GenAI For Developers | PDF | Artificial Intelligence https://www.scribd.com/document/887022171/GenAI-for-Developers
27. Understanding and Predicting Workloads for Improved ... https://www.researchgate.net/publication/320362605_Resource_Central_Understanding_and_Predicting_Workloads_for_Improved_Resource_Management_in_Large_Cloud_Platforms

28. Catalog en 04-07-2024 | PDF https://www.scribd.com/document/764732036/catalog-en-04-07-2024
29. Hideous Creatures - A Bestiary of The Cthulhu Mythos | PDF https://www.scribd.com/document/831655659/Hideous-Creatures-A-Bestiary-of-the-Cthulhu-Mythos
30. (PDF) The EU regional policy and ... https://www.researchgate.net/publication/385621923_The_EU_regional_policy_and_regional_policy_challenges_in_Czech_Republic_Hungary_Poland_Slovak_Republic_Montenegro_and_Serbia
31. cop-15-02-en.docx https://www.cbd.int/doc/c/b186/f870/fb3b25cbd874c7c3f5795bc3/cop-15-02-en.docx
32. Uzbekistan-2018-Public-Expenditure-and-Financial- ... https://documents1.worldbank.org/curated/en/869221561673501123/Uzbekistan-2018-Public-Expenditure-and-Financial-Accountability-PEFA-Performance-Assessment-Report.docx
33. Sustainable Development and Disaster Risk Reduction ... https://www.academia.edu/83288868/Sustainable_Development_and_Disaster_Risk_Reduction_Springer_Japan_2016_
34. Elevating Kubernetes Network Security Through Cilium ... https://www.scribd.com/document/973128419/Elevating-Kubernetes-Network-Security-Through-Cilium-Deployment
35. 531830WP0envir10Box345599B... https://documents1.worldbank.org/curated/en/652091468032338935/txt/531830WP0envir10Box345599B01PUBLIC1.txt
36. Cloud-Native Workload Orchestration at the Edge https://www.mdpi.com/1424-8220/23/4/2215
37. University of Verona https://core.ac.uk/download/604458527.pdf
38. Microservices-Based Architecture to Support Automation ... https://www.researchgate.net/publication/396410703_Microservices-Based_Architecture_to_Support_Automation_Applications_in_Smart_Industries_with_End-to-End_Real-Time_Requirements
39. Seamless Continuous Integration / Continuous Delivery (CI ... https://theses.hal.science/tel-05263254v1/file/these.pdf
40. (PDF) Cloud-Native Workload Orchestration at the Edge https://www.researchgate.net/publication/368580212_Cloud-Native_Workload_Orchestration_at_the_Edge_A_Deployment_Review_and_Future_Directions
41. A comprehensive survey on Software as a Service (SaaS) ... https://ieeexplore.ieee.org/iel7/6287639/6514899/10177956.pdf

42. (PDF) Virtualizing mixed-criticality systems: A survey on ... https://www.researchgate.net/publication/356822330_Virtualizing_mixed-criticality_systems_A_survey_on_industrial_trends_and_issues

43. glove.6B.100d.txt-vocab.txt https://worksheets.codalab.org/rest/bundles/0xadf98bb30a99476ab56ebff3e462d4fa/contents/blob/glove.6B.100d.txt-vocab.txt

44. Simple Index https://mirrors.sustech.edu.cn/pypi/simple/

45. wikilex-20070908-zh-en.txt - CMU School of Computer Science https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/Transfer/Chinese/wikilex-20070908-zh-en.txt

46. Contents https://ieeexplore.ieee.org/iel7/9969351/9969352/09969363.pdf

47. A Survey on Inference Engines for Large Language Models https://www.arxiv.org/pdf/2505.01658v1