Qwen

Sovereign Neural Network Governance: An Architectural Deep Dive into the Enforcement of RoH, Neurorights, and Non-Commercialization

This report provides a comprehensive analysis of the policy-enforcement mechanics within the sovereigntycore of the NeuroPC/OrganicCPU ecosystem. It focuses on the implementation-level details of how runtime and continuous integration (CI) systems guarantee adherence to four core governance invariants—Root of Humanity (RoH) risk ceiling of $\leqslant 0.3$, neurorights protections, stake-based authorization, and non-commercialization constraints—across the neural networking file families: `.nnetx`, `.nnetw`, and `.nnetq`. The analysis centers on the critical roles of two new binding and discovery artifacts, `.nnet-bind.aln` and `.nnetfs-index.aln`, which serve as the declarative contracts linking neural assets to their governing rules. The report deconstructs the multi-layered guard system, defines the precise ordering of checks executed by the sovereigntycore, and outlines the processes for model loading, execution, and lifecycle management. Furthermore, it offers specification-completeness insights and actionable guidance for developers implementing validators and integrating these mechanisms into their workflows, positioning robust enforcement as the foundational selling point for developer adoption.

The Binding and Discovery Framework: `.nnet-bind.aln` and `.nnetfs-index.aln`

The entire architecture for enforcing policies on neural networks is predicated on a clear separation of concerns: the *what* (the model's structure and parameters) is separated from the *rules* (its governance and operational constraints) . This principle is implemented through a dual-artifact framework consisting of the `.nnetfs-index.aln` and the `.nnet-bind.aln`. Together, they transform the problem of dynamic policy enforcement into one that can be significantly mitigated through static analysis and declarative wiring before any code is compiled or run. These artifacts provide a machine-

readable, authoritative map of all neural assets within a repository, enabling automated tools and the runtime kernel to validate compliance with the core governance invariants long before a model is loaded for inference or evolution. The `.nnetfs-index.aln` serves as the broad discovery map for the entire repository, while the `.nnet-bind.aln` acts as a granular deployment manifest, defining the contract between a specific software module and the neural network it intends to use.

The `.nnetfs-index.aln` is a per-repository index that functions as a master manifest for all neural-networking artifacts . Its primary role is to enable rapid, high-level discovery and validation by CI systems, linters, and AI-assistants . By providing a centralized, structured view of the entire neural asset space, it allows for cross-cutting validations that ensure consistency and integrity across the entire model ecosystem. The index is designed with several distinct sections, each cataloging a different type of artifact. The `models` section lists logical model identities (`model_id`), their kind (e.g., `encoder_decoder`, `ranker`), the path to their primary definition file (`.nnetx`), and associated metadata such as their RoH class and neurorights classification . For example, it would contain an entry for `bostrom-chat-gov-v1` pointing to `models/chat-gov-v1.nnetx` and marking it with a `low` RoH class and a `dreamsensitive` neurorights class . The `weights` section then links these models to their parameter files, specifying whether they are high-precision weights (`.nnetw`) or quantized weights (`.nnetq`), and noting if they are quantized . The `policies`, `caps`, and `bindings` sections provide pointers to the corresponding rule files for each model, such as `chat-gov-v1.nnet-policy.aln` and `chat-gov-v1.nnet-cap.aln` . Finally, the `logs` and `proofs` sections anchor the model to its evolution history and potential external proofs of ownership, respectively .

The true power of the `.nnetfs-index.aln` lies in its utility for static validation. An extended version of the `neuro-sovereign-lint` tool can be configured to consume this index and perform a series of automated checks against the global governance documents (`rohmodel.aln`, `neurorights.json`, etc.) . These lints enforce a set of critical invariants. First, they verify that every model listed in the `models` section has a corresponding entry in the `weights` section, ensuring no model exists without its parameters . Second, they confirm that all paths specified are local and confined within the repository, preventing dependencies on untrusted external sources and ensuring reproducibility . Third, and critically, they check for compliance with non-financial and neurorights constraints. If a model is marked with `nonfinancial: true` or has a `neurorights_class` of `dreamsensitive`, the linter will enforce that a corresponding `.nnet-proof.bchain.json` file must exist, anchoring the model to an immutable ledger like Googolswarm . This proactive validation catches a significant class of potential violations before they can even be committed to version control, drastically

reducing the attack surface and improving the developer workflow by failing fast on policy infractions. The index thus becomes the single source of truth for all AI-driven development tools, answering the fundamental question: "What models exist here, where are their components, what policies govern them, and where is their audit trail?" .

While the `.nnetfs-index.aln` provides a holistic view of the repository's contents, the `.nnet-bind.aln` provides the granular, per-module binding information necessary for runtime enforcement. It is essentially a "wiring slip" or a deployment manifest that declares exactly which module (e.g., a Rust crate or binary) is permitted to use which neural network under what specific constraints . Each row in this ALN file represents a binding, establishing a direct and verifiable link between a software component and a neural asset. The fields in this file are meticulously designed to encode the enforcement points of the sovereignty model directly into the deployment configuration. For instance, a binding for a chat assistant module might specify that it uses `bostrom-chat-gov-v1` . The binding explicitly names the paths to the model's architecture (`nnetx_path`), its weights (`nnetw_path`), and crucially, the paths to its dedicated policy (`policy_path`) and capability (`cap_path`) files . This explicit path resolution ensures that there is no ambiguity about which rules apply to which model at runtime.

Several fields in the `.nnet-bind.aln` are dedicated to encoding specific invariants. The `roh_ceiling` field allows for a per-binding RoH limit, which must be less than or equal to the global ceiling of 0.3 defined in the main `.rohmodel.aln` . This provides an additional layer of safety, allowing administrators to impose stricter limits on certain modules than the default. The `scope` field, with values like `public_noncommercial` or `internal_only`, is directly tied to the permissions defined in the project's `.stake.aln` and `.smart.json` files, ensuring that a module's operational domain aligns with its authorized token scope . Similarly, the `integration_depth` field guides the evaluation of proposals, requiring higher-risk integrations to be gated by EVOLVE tokens . Most importantly, the `.nnet-bind.aln` contains fine-grained capability flags like `suggest_only` and `readonly` . If `suggest_only` is true for a given binding, the associated module is programmatically forbidden from writing to the `.nnet-evolve.jsonl` stream for that model; it can only generate suggestions for changes, not enact them directly . This forces a separation between suggestion and action, routing all substantive modifications through the formal evolution pipeline. Likewise, the `readonly` flag prevents a module from mutating the model's state, restricting it to inference-only operations . By embedding these constraints directly into the binding manifest, the system makes policy compliance a matter of configuration and validation, rather than trusting the behavior of the module's code. The binding itself is then referenced from the sovereign kernel's main configuration, `bostrom-sovereign-kernel-v2.ndjson`, which instructs the `sovereigntycore` to load and validate this file on boot .

Field Name	Type	Purpose and Enforcement Role
`bind_id`	String	A stable, human-readable identifier for the binding, used in logs and ledgers to trace actions back to a specific deployment configuration .
`module_id`	String	The identity of the Rust crate or binary that this binding applies to (e.g., `neuro-assistant-api`). Used to attribute actions and enforce module-specific guards .
`model_id`	String	The logical identity of the neural network being bound (e.g., `bostrom-chat-gov-v1`). Must match the `model_id` in related policy and ledger files .
`nnetx_path`	Path	The relative path to the model's architecture and configuration file (`.nnetx`). This is the primary artifact to be loaded for inference .
`nnetw_path`	Path	The relative path to the model's high-precision weight file (`.nnetw`). This is the primary parameter file .
`policy_path`	Path	The relative path to the per-model risk and domain policy file (`.nnet-policy.aln`). Contains the RoH slice and allowed domains for this specific model .
`cap_path`	Path	The relative path to the per-model capability descriptor file (`.nnet-cap.aln`). Defines what the module is allowed to *do* with the network .
`roh_ceiling`	Float	A per-binding RoH ceiling, which must be validated against the global ceiling in `.rohmodel.aln` to ensure it does not exceed the system-wide limit .
`scope`	String	A tag (e.g., `public_noncommercial`) that determines the module's access to resources and its ability to propose risky changes, mapped to token scopes in `.stake.aln` .
`suggest_only`	Boolean	If true, the module is forbidden from creating evolution proposals (writing to `.evolve.jsonl`). It can only suggest changes, not act on them directly .
`readonly`	Boolean	If true, the module is restricted to read-only operations and cannot mutate the model's state or trigger evolutions .

In essence, the combination of `.nnetfs-index.aln` and `.nnet-bind.aln` creates a robust, declarative, and verifiable foundation for policy enforcement. The index provides the big picture for automated tools, while the binding file provides the ground-truth contract for runtime decisions. This architecture effectively shifts the burden of proof from runtime inspection of model blobs—which could hide malicious or non-compliant behavior—to the validation of these explicit, externally-bound governance artifacts. It ensures that every neural network is treated as a first-class, rights-governed object from the moment it is declared in a repository until it is actively used by a module . This approach directly addresses the research goal by making the enforcement of RoH, neurorights, and other constraints a transparent, codified, and automatable process.

Runtime Guard Pipeline: A Multi-Layered Defense System

The enforcement of governance invariants does not rely on a single, monolithic check but is instead orchestrated by a multi-layered guard pipeline executed by the `sovereigntycore` at critical junctures, primarily during system boot and when a module attempts to load or interact with a neural network. This pipeline is a deterministic sequence of validators that systematically verifies compliance with all core constraints before granting permission for any operation. The binding and discovery artifacts, `.nnet-bind.aln` and `.nnetfs-index.aln`, serve as the essential inputs that guide this pipeline, providing the resolver logic needed to connect a module's request to the specific set of policies, rights, and capabilities that apply to the target neural network. The overall design philosophy is one of fail-fast security: any step in the chain that fails results in an immediate refusal of the request, preventing non-compliant states from ever materializing .

The guard pipeline begins with the loading of the global governance artifacts. Before any module-specific bindings are processed, `sovereigntycore` loads the foundational stack of documents that define the project's legal and economic framework . This includes the global neurorights profile (`.neurorights.json`), the stakeholder roles and token scopes (`.stake.aln` and `.smart.json`), and the global RoH model definition (`.rohmodel.aln`) which enforces the universal ceiling of 0.3 . These files establish the baseline against which all subsequent, more granular checks are measured. They provide the ultimate authority for rules that apply system-wide, such as the prohibition on using neural data for employment or housing decisions . Having established this global context, the system proceeds to the next phase: resolving model-specific rules. When the kernel parses a `.nnet-bind.aln` file, it encounters a `model_id` for each binding . Using this ID, the resolver looks up the corresponding paths to the model's dedicated policy shard (`.nnet-policy.aln`), its neurorights profile (`.nnet-rights.json`), and its capability descriptor (`.nnet-cap.aln`) . This step is critical because it bridges the gap between a generic model identity and its specific, governed incarnation. It ensures that the checks applied to a model are tailored not just to the project's general rules but also to the particular risks and intended uses encoded in its own policy files.

A pivotal moment in the pipeline occurs after resolving the per-model artifacts. The system performs an intersection of capabilities and rights, applying a "strictest-wins" or logical AND principle to derive the effective permissions for a given module-network pair . For capabilities, the effective set is the intersection of the global capabilities defined in `.neuro-cap.aln` and the per-model capabilities in the file pointed to by `cap_path`

in `.nnet-bind.aln` . This means a module can only perform an action if both the global policy and the specific model's policy permit it. This prevents a module from gaining broader powers simply by binding to a specific model. Similarly, for neurorights, the effective rights profile is determined by taking the maximum (most restrictive) of the global rights and the per-network rights . If the global policy permits reading dream metrics but a specific model's `.nnet-rights.json` forbids it, the effective rule is to forbid it. This lattice-based approach ensures that model-specific constraints can never be overridden by a less strict global policy, reinforcing the principle that individual assets have their own, often stricter, governance .

With the global context, model-specific rules, and the derived effective permissions in hand, the pipeline executes a fixed, ordered sequence of guards. While the exact order may be subject to optimization, the documented flow provides a sound logical structure: stake → neurorights → RoH → envelopes → tokens → logging . This ordering is not arbitrary; it ensures that dependencies are resolved in a predictable and secure manner. The first guard, **Stake**, verifies that the entity attempting the action (identified by `subject_id`) has the necessary authorization according to the rules in `.stake.aln`. This could involve checking for a valid DID signature, multisig approval, or sufficient balance in a specific token scope . Without passing the stake check, no further checks are performed, as the actor is deemed unauthorized to participate in the governance process at all. Next, the **Neurorights** guard evaluates the proposed action against the effective neurorights profile. This guard would, for example, block any attempt to access data fields designated as sensitive or to use the model in a domain like employment, which is explicitly forbidden by the rights profile .

Following the stake and rights checks, the **RoH** guard is invoked. This is a multi-faceted check. First, it verifies that the operation does not cause the Root of Humanity metric to exceed the global ceiling of 0.3 . For a new model binding, this involves summing the weights in the model's `.nnet-policy.aln`'s `roh_slice` and ensuring it does not violate its allocated share from the global `.rohmodel.aln` . For an evolution proposal, it strictly enforces the monotonicity invariant: `roh_after` must be less than or equal to `roh_before` . This ensures that models can only become safer over time, a core tenet of the entire governance model. After the RoH check, the **Envelopes** guard inspects any bioscale or sensory data envelopes associated with the model. If the model's policy specifies `enforce_no_envelope_loosen == true`, this guard will verify that any referenced envelope shards satisfy a "tighten or equal, never loosen" constraint, preventing the degradation of safety boundaries around the user's inner signals . The fifth guard, **Tokens**, acts as an economic gatekeeper. It requires the proposing entity to spend tokens from a specific scope (e.g., `EVOLVE` tokens) to create an evolution proposal, as defined in the `.stake.aln` and `.smart.json` files . This introduces a cost to changing

the system's state, discouraging frivolous or malicious mutations. Finally, the **Logging** guard ensures that the decision is recorded immutably. All decisions, especially those involving model evolution, must be logged to the appropriate append-only streams, such as `.donutloop.aln` or its model-specific variant `.nnet-loop.aln`. This completes the cycle, providing a full audit trail for every action taken within the sovereign ecosystem.

Guard Stage	Description	Key Validation Checks
Stake	Verifies the actor's authorization based on roles, multisig, and DID signatures defined in `.stake.aln` and `.smart.json`.	Signature validation, DID registration, multisig threshold checks, token balance in the required scope.
Neurorights	Evaluates the proposed action against the effective neurorights profile (global + per-network).	Blocks access to forbidden data (e.g., dream metrics), usage in prohibited domains (employment, housing, etc.).
RoH	Ensures the Root of Humanity metric remains within safe bounds.	Sum of `roh_slice` weights ⩽ allocation; `roh_after` ⩽ `roh_before` for evolutions; `roh_after` ⩽ 0.3.
Envelopes	Checks constraints on bioscale and sensory data envelopes.	Validates that envelopes satisfy "tighten or equal, never loosen" if `enforce_no_envelope_loosen` is enabled.
Tokens	Requires spending tokens from a specific scope (e.g., EVOLVE) to initiate certain actions.	Deducts tokens from the proposer's account; validates the transaction is within their budget for that scope.
Logging	Ensures all decisions are recorded in immutable, append-only log files.	Writes a record to `.donutloop.aln` or `.nnet-loop.aln`; updates the hash chain correctly.

This layered, ordered guard system is the operational heart of the sovereignty model. It translates abstract, high-level invariants like "non-commercialization" or "RoH ⩽ 0.3" into concrete, executable checks against structured data. The binding artifacts are the indispensable bridge that connects these abstract rules to the concrete resources they are meant to govern. By designing a deterministic and fail-fast pipeline, the system ensures that policy enforcement is not an optional feature but a fundamental property of the architecture, making the ecosystem inherently resistant to accidental or deliberate violations of its core principles.

Lifecycle Management: Enforcing Invariants Through Evolution

Managing the evolution of a neural network is the most complex aspect of policy enforcement, as it involves permitting change while simultaneously ensuring that the core governance invariants are never compromised. The NeuroPC/OrganicCPU ecosystem

addresses this challenge with a specialized, append-only evolution stream and a dedicated, model-specific ledger, collectively forming a highly policed pathway for model modification. This process is designed to be transparent, auditable, and fundamentally aligned with the principles of monotone safety and stakeholder governance. Direct mutation of model files like `.nnetx` or `.nnetw` is forbidden; instead, any change must be proposed as a formal `EvolutionProposalRecord` in the `.nnet-evolve.jsonl` stream, which then flows through the same rigorous guard pipeline described previously . This mechanism transforms the concept of "updating a model" from an implicit, file-system-level operation into a formal, consent-based, and permanently logged event.

The `.nnet-evolve.jsonl` file serves as the dedicated EvolutionProposal stream for neural networks . It is an append-only NDJSON log where each line represents a single proposed change to a model. This structure is a specialization of the existing `.evolve.jsonl` schema, but with a fixed `kind` of `"NnetUpdate"` and a more specific `scope` field that delineates the nature of the change, such as `architecture`, `hyperparams`, `training_data`, or `checkpoint` . Each proposal record contains critical metadata for enforcement. It includes the `proposal_id`, the `model_id` of the network to be changed, and the `subject_id` of the entity making the proposal . Crucially, it must contain the `roh_before` and `roh_after` values, which are the anchors for the system's most important safety constraint: RoH monotonicity . The invariant $roh_{after} \leq roh_{before}$ and $roh_{after} \leq 0.30$ must be satisfied for the proposal to pass the RoH guard . This ensures that a model can only evolve into a state that is either equally risky or less risky than its previous state, preventing regressions in safety. The record also includes cryptographic hashes (`arch_hash_before`, `params_hash_after`, etc.), which point to snapshots of the model's artifacts (`.nnetx` files or config ALNs) rather than the raw weight files themselves, ensuring that the proposal refers to verifiable content .

Once a proposal is submitted, it enters the decision-making phase, which is tracked by the `.nnet-loop.aln`. This donutloop-style ledger is a model-specific view of the evolution history, analogous to the global `.donutloop.aln` but scoped exclusively to the lifecycle of a single neural network . Each row in this ledger corresponds to a decision made on a proposal from the `.nnet-evolve.jsonl` stream. The ledger's entries are cryptographically chained, with each row's `hexstamp` depending on the previous row's `prev_hexstamp`, creating an immutable and append-only timeline . A typical entry in the `.nnet-loop.aln` would include the `entry_id`, the `model_id`, the `proposal_id` it resolves, the final `decision` (which must be one of `{Allowed, Rejected, Deferred, Genesis}`), the RoH values, and pointers to the old and new `.nnetx` snapshots if the decision was `Allowed` . This creates a definitive, auditable record of a model's entire evolutionary path. Any module attempting to load a model would consult

this ledger to understand its lineage and ensure it is operating on an approved state. The existence of this dedicated ledger simplifies auditing immensely, as developers and auditors no longer need to scan the entire system's evolution log to trace a specific model's history; they can simply inspect the relevant `.nnet-loop.aln` file .

For an extra layer of security, authenticity, and proof-of-ownership, particularly for high-value or critical models, the evolution process supports on-chain anchoring. This is facilitated by the `.nnet-proof.bchain.json` artifact, an envelope that wraps a snapshot of key artifacts and their corresponding ledger entry and anchors it to an external blockchain like Googolswarm or Organicchain . This proof contains the hashes of the model's `.nnetx` snapshot, the `donut_entry_id` from the `.nnet-loop.aln`, and the transaction hash from the blockchain, along with multisig approval signatures . This creates a cross-chain audit trail that proves the existence and state of the neural asset at a specific point in time, immune from alteration by any party within the NeuroPC/OrganicCPU ecosystem alone. The `nonfinancial: true` and `soulnontradeable: true` flags from the neurorights profile are propagated into this proof, ensuring that even on-chain, the asset is recognized as a sovereign, non-commercial entity . This mechanism is entirely optional but provides powerful guarantees for models that require public trust or are subject to intellectual property claims within the sovereign framework.

The entire evolution lifecycle is therefore heavily policed to ensure invariant compliance. The process begins with a stake-gated proposal: only entities with the `EVOLVE` token scope can create a new line in the `.nnet-evolve.jsonl` file . This proposal then flows through the standard guard pipeline. During the RoH check, the `roh_after <= roh_before` invariant is rigorously enforced, which is a hard constraint built into the validation logic of the `EvolutionProposalRecord` . The stake guard ensures that only authorized stakeholders can initiate changes, while the token guard requires them to pay a fee, introducing an economic disincentive for spam or destructive proposals. The logging guard ensures that the outcome of the evaluation (whether the proposal is `Allowed` or `Rejected`) is written to the `.nnet-loop.aln` ledger, updating the model's canonical state or rejecting the change . If a proposal is `Allowed`, the ledger is updated, and the module's configuration can be safely reloaded to point to the new `.nnetx` snapshot. This entire process—from proposal to logging—is a closed loop of checks and balances, ensuring that every change to a neural network is intentional, consented to by the proper authorities, economically accounted for, and permanently recorded. It embodies the principle that evolution is not freedom from rules, but evolution *under* rules.

Capability and Rights Interpolation: Defining Module Behavior

The behavior and permissible actions of a software module that interacts with a neural network are not defined by its source code alone but are explicitly constrained by a combination of globally defined capabilities and model-specific rights. The system employs a sophisticated interpolation mechanism to derive an effective capability set and an effective rights profile for each module-network pair. This process ensures that the least privilege principle is upheld and that neurorights can be made stricter on a case-by-case basis, preventing a module from escaping its governance constraints simply by loading a specific model. The primary artifacts driving this interpolation are the global capability descriptor, `.neuro-cap.aln`, and the per-model capability descriptor, `.nnet-cap.aln`, alongside the global neurorights profile, `.neurorights.json`, and the per-model neurorights profile, `.nnet-rights.json`.

The capability system is designed around the concept of intersection. When a module, identified by its `module_id`, binds to a neural network via an entry in `.nnet-bind.aln`, the `sovereigntycore` does not simply accept the capabilities listed in the binding's `cap_path`. Instead, it performs a logical AND operation between two sets of capabilities. The first set is derived from the global `.neuro-cap.aln`, which defines a baseline set of capabilities applicable to all neuromodulation-related activities within the project . The second set comes from the per-model `.nnet-cap.aln`, which can add or, more importantly, restrict capabilities for a specific network . For example, the global `.neuro-cap.aln` might grant a module the `may_propose_evolve` capability. However, if the per-model `.nnet-cap.aln` for the network it binds to explicitly sets `may_propose_evolve: false`, the effective capability for that module-network pair is `false`. This prevents a module from gaining broader powers by connecting to a model that has a more restrictive policy. This intersection is performed at runtime during the bootstrapping phase, where the kernel validates each binding in `.nnet-bind.aln` against the resolved `.nnet-cap.aln` file .

The `.nnet-cap.aln` file provides a rich vocabulary of capabilities that precisely define a module's potential interactions with a model. The `capabilities` section contains boolean flags for common actions, such as `suggest_only`, `never_actuate`, and `may_read_metrics` . The `domains` section further refines these capabilities by specifying the depth of integration allowed for different functional areas, such as `language_cowriter` or `biofeedback_assistant` . This domain-based scoping allows for fine-grained control, ensuring a module assigned to `biofeedback_assistant` cannot, for instance, write to the live system configuration,

even if it had the technical ability to do so. The `guards` section of the `.nnet-cap.aln` itself acts as a declaration of what is required for the module to operate, listing dependencies like `stake_guard_required: true` and `neurorights_guard_required: true` . This creates a self-documenting contract that clearly articulates the prerequisites for the module's functionality. By intersecting these per-model capabilities with the global baseline, the system ensures that a module's operational envelope is always the smallest possible set that satisfies both the general project policy and the specific constraints of the model it is interacting with.

Similarly, the neurorights system employs a "strictest-wins" interpolation model. At boot time, the `sovereigntycore` constructs an effective neurorights profile for each binding by comparing the globally defined rights in `.neurorights.json` with the per-model rights in the file specified by `policy_path` in `.nnet-bind.aln` . This is not a simple override but a field-by-field maximum operation based on a predefined lattice of rights severity. For example, if the global `.neurorights.json` has `dream_state_sensitive: false` but a specific model's `.nnet-rights.json` has `dream_state_sensitive: true`, the effective right for that model is `true` . This ensures that a model can introduce stricter privacy or sensitivity constraints than the project default, but it can never relax them. This is critical for protecting users, as it allows for the creation of specialized, high-sensitivity models without compromising the neurorights of users who interact with them. Other fields like `noncommercial`, `soulnontradeable`, and the arrays in `forbid_decision_use` and `allowed_uses` are similarly handled . The `forbid_decision_use` array, for instance, would be the union of the global list and the model-specific list, making the ban on discriminatory use even stronger at the model level. The `right_to_forget` SLA is also inherited, with a model potentially specifying a shorter, more stringent deadline than the global default .

The following table illustrates how these profiles are interpolated to form the effective rights for a module-network binding.

Effective Right Field	Global Profile Source	Per-Model Profile Source	Interpolation Logic
`mental_privacy`	`.neurorights.json`	`.nnet-rights.json`	Max(global, model)
`dream_state_sensitive`	`.neurorights.json`	`.nnet-rights.json`	Max(global, model)
`noncommercial`	`.neurorights.json`	`.nnet-rights.json`	Max(global, model)
`soulnontradeable`	`.neurorights.json`	`.nnet-rights.json`	Max(global, model)
`forbid_decision_use`	`.neurorights.json`	`.nnet-rights.json`	Union(global_array, model_array)
`data_scope.may_read_neural_patterns`	`.neurorights.json`	`.nnet-rights.json`	Max(global, model)
`right_to_forget.forget_sla_hours`	`.neurorights.json`	`.nnet-rights.json`	Min(global_hours, model_hours)

This dual-interpolation system—intersection for capabilities and a strictest-wins model for rights—provides a powerful and flexible yet secure framework for governing module behavior. It decouples the definition of a module's potential actions from the definition of a model's inherent properties. The global files provide a stable, project-wide baseline, while the per-model files allow for specialization and stricter controls where necessary. This ensures that the runtime enforcement logic is both predictable and robust. A developer can understand a module's behavior by examining its binding, which points to both a global and a specific policy, knowing that the final, effective permissions are a guaranteed superset of requirements and a guaranteed subset of powers. This clarity is a key part of the system's "selling point," as it makes the governance model transparent and auditable, building trust that the system will not permit actions that violate its core principles.

Specification and Implementation Guidance for Developers

To translate the architectural vision into a functional system, developers working within the `organiccpualn` and `sovereigntycore` crates must implement concrete Rust types and validators that correspond to the newly defined ALN and JSON artifacts. This section provides a detailed specification for these implementations, focusing on the structures for `.nnet-bind.aln` and `.nnetfs-index.aln`, and outlining the necessary validator traits. The guiding principle is to make the policy enforcement logic explicit, testable, and tightly integrated with the build and runtime environments. The provided ALN schemas serve as the direct blueprint for these Rust structs, leveraging the `serde` crate for parsing and serialization .

First, the core data structures must be defined. For the `.nnet-bind.aln` file, a `BindingRow` struct would encapsulate the information for a single model-to-module connection. This struct would mirror the columns in the ALN file, with fields for `bind_id`, `module_id`, `model_id`, `nnetx_path`, `nnetw_path`, `policy_path`, `cap_path`, `roh_ceiling`, `scope`, `suggest_only`, and `readonly` . Similarly, a top-level `NnetBindIndex` struct would contain a vector of these `BindingRow` instances, representing the entire binding table. Analogously, the `.nnetfs-index.aln` would be modeled by a `NnetFsIndex` struct containing separate vectors for `models`, `weights`, `policies`, `caps`, `bindings`, `logs`, and `proofs`, with each vector holding a dedicated struct for its respective ALN section (e.g., a `ModelIndexEntry` struct) . These structs, when combined with `serde` attributes, will automatically handle the parsing of the CSV-like ALN format into native Rust objects.

With the data structures defined, the next step is to implement validation logic. This is achieved by defining a `Validator` trait, perhaps named `Validate`, that has a method `fn validate(&self) -> Result<(), ValidationError>`. Each of the major struct types (`NnetBindIndex`, `NnetFsIndex`, `NnetPolicy`, etc.) would then implement this trait. This modular approach allows for focused, unit-testable validation logic. For the `BindingRow` struct, the `validate` implementation would perform several key checks. It would first ensure the `roh_ceiling` is a valid float and is less than or equal to the global RoH ceiling of 0.3, which could be loaded from the project's `.rohmodel.aln` at the time of validation . It would then check the `scope` field against a predefined enum of valid scopes (e.g., `PublicNoncommercial`, `InternalOnly`) and cross-reference it with the token scopes defined in the project's `.smart.json` file to ensure compatibility . The `suggest_only` flag would be validated against the capabilities in the file pointed to by `cap_path`; if `suggest_only` is true, the validator would recursively load and validate that the corresponding `.nnet-cap.aln` does not grant the `may_propose_evolve` capability . Finally, it would perform filesystem checks to ensure that all paths listed (e.g., `nnetx_path`, `policy_path`) are relative and resolve to existing files within the repository's root directory, preventing path traversal attacks or reliance on external, untrusted artifacts .

The `.nnetfs-index.aln` validator would operate at a higher level of aggregation. Its `validate` method would iterate through the indexed artifacts and perform cross-referencing checks. It would verify that for every `model_id` in the `models` section, there is a corresponding entry in the `weights`, `policies`, and `caps` sections . It would ensure that every path listed in the `bindings` section of the index points to a valid `.nnet-bind.aln` file and that loading and validating that file passes all its own checks. A critical lint would be to check for `nonfinancial` or `dreamsensitive` classifications. If a model is marked with either, the validator would confirm that there is

a corresponding entry in the `proofs` section and that the linked `.nnet-proof.bchain.json` file contains the mandatory `nonfinancial: true` and `soulnontradeable: true` fields, ensuring that high-sensitivity assets are properly anchored . This high-level validation can be run as a pre-commit hook or a CI gate, providing a powerful defense-in-depth strategy.

Integrating these validators into the development workflow is crucial. The primary mechanism for enforcement is through the CI/CD pipeline. A dedicated CI job should be created that runs the `neuro-sovereign-lint` tool. This tool would take the path to a repository as input, locate its `.nnetfs-index.aln`, and invoke the `validate` methods on the parsed index and all artifacts it references . Any failure in this validation process should cause the CI job to fail, blocking the merge of any pull request that introduces non-compliant neural assets. This makes policy enforcement an automated, unavoidable part of the development lifecycle. Developer-facing documentation and examples should emphasize this "fail-fast" philosophy, showing how to use the linter locally to catch errors before committing code.

Furthermore, the AI-assistant and developer experience must be guided towards compliant practices. When an AI assistant is asked to modify a model, it should not be given direct access to the file system to edit `.nnetx` or `.nnet-cap.aln` files. Instead, its output should be formatted as a patch or a complete replacement for a specific artifact. The assistant's prompt engineering and tooling should be designed to help the developer craft a valid `EvolutionProposalRecord` in the `.nnet-evolve.jsonl` stream . For instance, if a developer wants to change a model's hyperparameters, the AI should generate the correct JSON structure for an `NnetUpdate` proposal, including the `roh_before` and `roh_after` values, and suggest the appropriate `scope` (e.g., `hyperparams`). This redirects the developer's intent away from direct mutation and towards the formal, auditable evolution process. The `.nnetfs-index.aln` serves as the single lookup for all such tools, telling them what models exist, what their current configurations are (by referencing the `nnet-loop.aln`), and what the rules for changing them are. By making the evolution stream the canonical and only sanctioned way to alter a model's state, the system ensures that every change is captured, reviewed, and logged, fulfilling the promise of a truly sovereign and accountable neural network ecosystem.

Synthesis of Enforcement Mechanics and Architectural Integrity

The proposed architecture for policy enforcement within the NeuroPC/OrganicCPU ecosystem demonstrates a high degree of completeness and internal consistency in addressing the core research goal. It successfully establishes a robust, multi-layered defense system that guarantees compliance with the four critical governance invariants—RoH $\leqslant$ 0.3, neurorights protections, stake-based authorization, and non-commercialization constraints—across the neural network file formats (.nnetx, .nnetw, .nnetq). The strength of this design lies not in a single, monolithic enforcement mechanism, but in the synergistic interplay between declarative artifacts, automated validation, and a deterministic runtime guard pipeline. This synthesis confirms that the architecture is not merely theoretical but provides a concrete, implementable blueprint for building a trustworthy and sovereign neural network platform.

The foundational principle of explicit, external binding is the cornerstone of this entire system. By separating the model's content (`.nnetx`, weights) from its governance rules (`.nnet-policy.aln`, `.nnet-rights.json`, `.nnet-cap.aln`) and declaring the connections between them in binding artifacts (`.nnet-bind.aln`, `.nnetfs-index.aln`), the architecture makes policy enforcement a verifiable and automatable process . The `.nnetfs-index.aln` provides a comprehensive discovery map that enables powerful static analysis via CI linters, catching a vast majority of potential violations before code is ever compiled or run . This proactive validation forms the first line of defense. The `.nnet-bind.aln` then acts as the ground-truth contract for runtime operations, translating abstract invariants into concrete, path-based checks that `sovereigntycore` can execute with certainty .

The runtime guard pipeline is the operational engine of this architecture. Its fixed, ordered sequence of checks—stake, neurorights, RoH, envelopes, tokens, and logging—ensures that every action is vetted against the full spectrum of governance rules in a predictable and secure fashion . The interception of capabilities and the "strictest-wins" interpolation of neurorights demonstrate a sophisticated understanding of security principles, preventing privilege escalation and ensuring that model-specific constraints can only tighten, never loosen, the protections afforded to users . This pipeline directly enforces the core invariants: the RoH guard polices the `roh_after <= roh_before` invariant, the neurorights guard enforces the prohibitions defined in the rights profiles, the stake guard ensures only authorized actors can propose changes, and the non-

commercialization flag is respected across all layers, from binding validation to on-chain proof anchoring .

The lifecycle management system for neural networks is a particularly strong component of the design. By mandating that all changes flow through the append-only `.nnet-evolve.jsonl` stream and be recorded in the model-specific `.nnet-loop.aln` ledger, the system provides an immutable and auditable history for every network . This process is itself policed by the same guard pipeline, ensuring that evolution is not an escape hatch from governance but a governed extension of it. The optional but powerful on-chain anchoring via `.nnet-proof.bchain.json` adds a final layer of external, third-party verification, cementing the sovereignty of the assets .

In summary, the architectural specification is highly complete. Every requirement from the research goal is addressed:

- **RoH ⩽ 0.3:** Enforced globally via `.rohmodel.aln` and per-model via `.nnet-policy.aln`, checked by the runtime guard pipeline and the evolution process.
- **Neurorights Protections:** Defined at the project level in `.neurorights.json` and interpolated with stricter per-model rules in `.nnet-rights.json`, enforced by the neurorights guard and capability restrictions.
- **Stake Authorization:** Managed through `.stake.aln` and `.smart.json`, gating access to the system and the ability to create evolution proposals.
- **Non-Commercialization:** Encoded as a hard constraint in neurorights profiles, which are respected throughout the binding, runtime, and evolution processes.

The primary recommendation for implementation is to translate the provided ALN schemas into well-defined Rust types within `organiccpualn` and `sovereigntycore`, and to rigorously implement the validation logic for each, turning the declarative rules into executable code. The enforcement fidelity demonstrated by this architecture should be the primary "selling point" for developers, as it provides a tangible guarantee of safety, accountability, and adherence to the project's ethical and legal commitments.

Reference

1. A Neuroadaptive Blueprint for Non-Invasive Vision ... https://www.researchgate.net/publication/

390345115_Through_the_Ear_We_See_A_Neuroadaptive_Blueprint_for_Non-Invasive_Vision_Restoration_via_Auditory_Interfaces_Proposed_Table_of_Contents_Part_I_Foundations_of_Sensory_Rerouting

2. (PDF) Analysis of Workloads for Cloud Services https://www.academia.edu/96340801/Analysis_of_Workloads_for_Cloud_Services