Qwen

From Proposal to Proof: Verifying System Integrity Through NeuroPC's Sovereign Guard Pipeline and Embedded Shard V

The Guard Pipeline as Constitutional Enforcer

The architectural foundation of NeuroPC's security and integrity rests upon a meticulously defined and sequentially enforced guard pipeline, orchestrated exclusively by the `sovereigntycore` component . This pipeline transcends the conventional notion of a validation step; it functions as the primary mechanism for enforcing the system's constitutional invariants, translating abstract principles encoded in the sovereign data plane into concrete actions that govern all state changes . Every modification to the system's real state—be it an update to the RoH model, a change in stake distribution, or an alteration of lifeforce envelopes—must navigate this gauntlet before any shard mutation occurs . The entire design embodies a zero-trust philosophy where no entity, including powerful client applications like Jupyter notebooks, is granted implicit privilege [9] [22]. Instead, every action must be formally proposed and rigorously evaluated against a fixed hierarchy of rules. This section provides a technical deep dive into the mechanics of this pipeline, analyzing its ordered structure, the distinct function of each guard stage, and the central role of `sovereigntycore` as the kernel of authority.

The specified enforcement logic follows a strict, non-negotiable sequence: **RoH guard → neurorights guard → stake/token guard → donutloop logging** . This order is not arbitrary but represents a deliberate policy hierarchy, ensuring that more fundamental constraints are evaluated before less critical ones. The first stage, the RoH guard, acts as the ultimate line of defense, mandating that any proposed change must not result in an increase of the Risk of Harm metric above the constitutional limit of 0.3 . This constraint appears to be absolute, forming a prerequisite so foundational that no other consideration can supersede it. A proposal that fails this initial check is immediately rejected without proceeding to subsequent stages, effectively preventing any potential for existential risk from being introduced into the system. This immediate termination on failure reinforces the principle that physical and cognitive safety is paramount.

Following the RoH guard, the neurorights guard applies a second layer of policy, rooted in legal and ethical frameworks . This stage enforces policies such as pre-access checks,

non-commercial use clauses, and procedures for Over-the-Air (OTA) updates [1]. Its position after the RoH guard suggests a secondary priority: while not as immediately existential as RoH, the protection of neurorights is considered a core tenet of the system's constitution. This guard ensures that even if a change is deemed safe from a risk-of-harm perspective, it must still comply with the established rights and permissions governing the dream states and associated data [1]. For example, a proposal to alter a neural model might be technically safe but would fail at the neurorights guard if it involved data subject to non-commercial clauses or required access that has not been properly authorized.

The third stage, the stake/token guard, introduces the economic and governance dimensions of the system. This guard verifies two key pieces of information: the proposer's token balance and their assigned role . It checks whether the proposing entity holds sufficient `EVOLVE` or `SMART` tokens to cover the cost of the proposed change and whether their specific role (e.g., Host, OrganicCPU, ResearchAgent) grants them the necessary scope to request that particular type of modification . This stage serves as the gatekeeper for resource allocation and permission management. Only entities with the requisite economic stake and delegated authority can advance past this point. The placement of this guard after the safety and rights-based checks ensures that governance is only applied to proposals that have already passed more fundamental scrutiny.

Finally, the donutloop logging stage operates differently from the preceding guards. It is not an active filter but a passive, append-only audit log . Once a proposal has successfully passed through all three preceding gates and received an "Allowed" verdict, `sovereigntycore` appends a new, hash-chained entry to the `.donutloop.aln` shard . This entry records the decision, the hash of the original proposal, and the context of the evaluation. Its purpose is to create an immutable and cryptographically linked history of all decisions made by the guard pipeline. By placing this stage last, the architecture ensures that the log captures the final outcome of the complete evaluation process, providing an auditable proof that the enforcement logic was correctly applied. This transforms the donutloop from a simple log into a tamper-evident ledger that serves as the canonical record of system evolution .

At the heart of this entire enforcement apparatus is the `sovereigntycore` Rust crate, which functions as the singular, authoritative executor of the guard pipeline . It is the only component permitted to interpret the `.evolve.jsonl` stream and mutate the protected sovereign shards . All other parts of the system, including background daemons, CLI tools, and Jupyter notebook kernels, must route their requests for state change through `sovereigntycore`. This centralized control prevents any form of direct manipulation of the system's real state, effectively creating a hard boundary around the

sovereign data plane. When a client submits an `EvolutionProposalRecord`, it is not directly writing to `.rohmodel.aln` or `.stake.aln`; instead, it is appending a request to the `.evolve.jsonl` stream for `sovereigntycore` to process . The crate then loads the relevant shards, as specified in the sovereign manifest, and executes the full, ordered sequence of guards. Only upon receiving a successful verdict does it proceed to perform the actual shard mutations and update the donutloop ledger, often in an atomic fashion to ensure consistency . This design makes `sovereigntycore` the true kernel of the NeuroPC operating system, responsible for upholding the constitutional order.

The operational proving ground for this enforcement logic is the lifecycle of an `EvolutionProposalRecord`. This lifecycle begins with a client application, such as a Jupyter notebook, drafting a proposal. This involves constructing a candidate change in memory, often by simulating tweaks to parameters like RoH weights or envelope bounds to observe potential effects without committing to them . Once a researcher determines that a change is promising, the notebook serializes this proposal according to the precise schema of the `.evolve.jsonl` shard and submits it via a binding to the `sovereigntycore` crate . The submission triggers the evaluation phase, where `sovereigntycore` takes over. It reads the proposal, loads the necessary typed Rust structs derived from the corresponding shard schemas (e.g., `.rohmodel.aln`, `.stake.aln`, `.neurorights.json`), and runs the ordered guard pipeline against the proposal's declared scope, expected RoH delta, and requested resources . The evaluation yields one of three possible outcomes: Allowed, Rejected, or Deferred.

If the verdict is "Allowed," `sovereigntycore` proceeds to execute the change. This involves performing the targeted shard mutations—for instance, updating the feature space and weights in `.rohmodel.aln` or altering address bindings in `.stake.aln`—and then atomically appending a new entry to the `.donutloop.aln` ledger . This atomic operation ensures that either both the state change and the audit log entry succeed, or neither does, maintaining the integrity of the system's history. If the verdict is "Rejected" or "Deferred," the proposed changes are discarded, and the reason for the decision is recorded in the donutloop log . This entire flow demonstrates a critical principle: actuation is entirely conditional upon evaluation by the centralized guard. The donutloop ledger becomes the ultimate, verifiable source of truth for what changes were actually implemented, providing an immutable record that proves the enforcement logic was faithfully executed. This closed-loop system, where every potential state transition must be vetted by `sovereigntycore` before taking effect, is the cornerstone of NeuroPC's claim to single-plane sovereignty.

Guard Stage	Primary Function	Key Constraints Enforced	Position in Pipeline
RoH Guard	First-line safety enforcement	Risk of Harm (RoH) must remain ⩽ 0.3	1
Neurorights Guard	Legal and ethical policy enforcement	Pre-access checks, non-commercial clauses, OTA update protocols	2
Stake/Token Guard	Governance and economic validation	Valid token balance (`EVOLVE`/`SMART`), role-based scope permissions	3
Donutloop Logging	Immutable audit trail creation	Records decision (Allowed/Rejected/Deferred) and hash-linked context	4

This structured, multi-stage enforcement process creates a robust defense-in-depth strategy. A malicious or erroneous proposal cannot bypass a lower-priority constraint to satisfy a higher one. For example, a proposal originating from an entity with insufficient tokens or improper scope would fail at the stake/token guard, regardless of its RoH profile or compliance with neurorights. Similarly, a change that violates the non-commercial clause would be caught by the neurorights guard, even if it passed the RoH and stake checks. This layered approach ensures that the system's integrity is maintained across multiple axes of concern: safety, ethics, and governance. The `sovereigntycore` crate, guided by the constitutional NDJSON manifest, acts as the impartial judge, applying these rules consistently and without exception, thereby transforming the NeuroPC architecture from a mere collection of components into a self-governing, rule-abiding entity.

Shard Schemas and Formal Verification as Embedded Invariants

A defining characteristic of NeuroPC's single-plane sovereignty model is the principle that the data plane is also the verification plane . In this architecture, invariants are not merely hardcoded rules within the `sovereigntycore` logic; they are properties explicitly defined and embedded within the structure and content of the system's constituent shards. This approach elevates the shards from passive storage containers to active, declarative contracts that govern their own integrity. Each shard type, identified by its extension (e.g., `.aln`, `.json`, `.jsonl`), is treated as a first-class legal object with its own schema, cross-file invariants, and associated verification artifacts . This section analyzes how formal schemas, continuous integration (CI) invariants, and mathematical viability proofs are woven directly into the fabric of the sovereign data plane, creating a

multi-layered system of embedded verification that complements the guard pipeline's runtime enforcement.

Each shard in the sovereign data plane is associated with a formal schema that defines its precise structure, data types, and constraints . These schemas act as machine-readable contracts, specifying exactly what constitutes valid data for each shard. For instance, the `.rohmodel.aln` shard is governed by a schema that enforces rules such as "RoH $\leqslant$ 0.3," "non-negative weights," and "monotone envelopes" . Similarly, the `.stake.aln` shard's schema would define the structure for roles, multisig rules, and address bindings . While the provided materials do not specify the exact schema language used (e.g., JSON Schema [1], Protocol Buffers, or a custom format), their existence is fundamental to the architecture's integrity. The use of typed Rust structs, generated from these schemas, provides strong compile-time guarantees about data integrity, preventing malformed proposals from ever reaching the evaluation stage of the guard pipeline . This schema-driven approach allows policies themselves to be treated as data, enabling them to be updated through the same evolutionary process they are designed to govern, provided the update proposal passes the guard checks.

Beyond runtime validation, the system incorporates CI-enforced type checks as a crucial pre-evaluation barrier . This practice extends formal verification beyond the live system into the development and deployment pipeline. Before a proposal can be submitted to the `.evolve.jsonl` stream, it likely undergoes a series of automated checks performed by a Continuous Integration (CI) system. These checks could include:

- **Schema Validation:** Ensuring that any new data intended for a shard conforms precisely to its designated schema, catching structural errors early.
- **Code Linter and Formatter:** Enforcing coding standards for any Rust code associated with the proposal, promoting consistency and reducing bugs.
- **Static Analysis:** Performing deeper analysis of proposed code changes to identify potential logical flaws or security vulnerabilities.
- **Formal Property Checking:** Using specialized tools to verify that proposed modifications to the Tsafe/viability kernel geometry adhere to their underlying mathematical proofs .

This CI layer acts as a first-pass filter, catching errors and inconsistencies before they ever touch the live system. This improves efficiency by reducing the load on the guard pipeline and enhances overall system stability by preventing flawed proposals from entering the evolution stream altogether . It represents a commitment to quality assurance as a foundational element of the system's sovereignty.

Perhaps the most sophisticated layer of embedded verification is the integration of Tsafe/viability proofs, which are stored within the `.vkernel.aln` shard . This shard contains the Tsafe/viability kernel geometry and envelopes, which mathematically define the set of all allowed system states and actions . This concept is analogous to formal methods in computer science and control theory, where a system's behavior is guaranteed to remain within a predefined "safe set" [16] [19] . The inclusion of these proofs elevates the system's guarantees from heuristic rule-following to mathematical certainty. The `sovereigntycore` guard pipeline almost certainly includes a dedicated check that verifies the proposed change maintains the system's membership in this viable set. This check would utilize the geometric information from the `.vkernel.aln` shard to perform a formal verification that the proposed evolution does not push the system into a region of unbounded or undesirable behavior. A failed viability proof would cause the proposal to be rejected, irrespective of its status on the RoH, neurorights, or stake checks. This mechanism provides a powerful, non-heuristic guarantee of long-term system stability and safety, moving beyond simple thresholds to encompass the global dynamics of the system's state space.

The table below outlines the key sovereign shards, their purpose, and the embedded verification mechanisms they employ.

Shard File	Purpose	Associated Verification Mechanisms
`.rohmodel.aln`	Stores the RoH feature space, model weights, and associated invariants .	Formal schema enforcing RoH $\leq$ 0.3, non-negative weights, monotone envelopes .
`.stake.aln`	Defines roles, multisig rules, EVOLVE/SMART scopes, and address bindings .	Formal schema for structuring roles, addresses, and multisig configurations .
`.neurorights.json`	Contains neurorights flags and enforcement modes (pre-access, OTA, donutloop logging, non-commercial clauses) .	Formal schema defining right types, modes, and associated metadata [1] .
`.vkernel.aln`	Holds the Tsafe/viability kernel geometry and envelopes for allowed state/action sets .	Embedded Tsafe/viability proofs encoding mathematical guarantees of system stability .
`.biosession.aln`	Stores per-session BioState summaries and RoH-relevant traces .	Formal schema for time-series BioState data and RoH-related metrics .
`.donutloop.aln`	Functions as a hash-chained audit log of accepted/rejected/deferred proposals and guard decisions .	Cryptographic hashing linking entries to provide an immutable, tamper-evident history .

By embedding these diverse forms of verification directly into the shards, the NeuroPC architecture achieves a high degree of self-consistency and resilience. The schemas provide a static, structural contract. The CI invariants offer a dynamic, pre-deployment validation layer. And the Tsafe/viability proofs supply a mathematical, dynamical guarantee. Together, they create a comprehensive verification stack that works in concert with the guard pipeline. The guard pipeline acts as the runtime enforcer, checking proposals against the current state of the system as described by these verified shards.

This symbiotic relationship between the data plane (containing the definitions) and the execution plane (`sovereigntycore`, containing the rules) is what enables the single-plane sovereignty model to function as a coherent and trustworthy system. It ensures that the system's state always remains consistent with its foundational principles, making the data itself the ultimate arbiter of its own validity.

The Anchor-Trio Crates as the Unifying API Surface

In the NeuroPC architecture, the principle of treating everything as a client extends to powerful interactive environments like Jupyter notebooks, which are designed not as privileged runtimes but as sandboxed clients of the sovereign data/guardrail plane . The integrity of this model hinges on a carefully constrained Application Programming Interface (API) surface that mediates all interactions with the system's real state. This API is embodied by a trio of specialized Rust crates known collectively as the "anchor-trio": `organiccpucore`, `organiccpualn`, and `sovereigntycore` . These crates are not merely utility libraries; they are the fundamental building blocks that translate the abstract concepts of the single-plane sovereignty model into concrete, verifiable programmatic interfaces. They form the exclusive gateway through which any client, including a Jupyter kernel, can read system data or propose changes, thereby enforcing the critical separation between analysis/drafting and actuation.

The `organiccpualn` crate serves as the primary interface for reading data from the sovereign shards . It provides typed Rust bindings that map directly to the formal schemas of the various shard types, such as `.rohmodel.aln`, `.stake.aln`, `.neurorights.json`, and others . When a client needs to inspect the current state of the system—for example, to plot the latest RoH trends or review stakeholder roles—it must use the functions exposed by `organiccpualn`. This crate is responsible for deserializing the ALN or JSON data from the shards into strongly-typed Rust structures. This mechanism provides several critical safety benefits. First, it enforces schema compliance at the boundary, preventing clients from attempting to interpret malformed or unexpected data. Second, it abstracts away the physical details of the shard files (e.g., their paths and binary vs. text formats), presenting a clean, high-level view of the system's state. Third, by limiting read access to these specific, well-defined functions, it prevents clients from bypassing the intended data views and accessing raw files or process-global state, thus preserving the integrity of the data plane .

The `organiccpucore` crate provides another essential layer of the API, exposing functions related to the core bio-cognitive state of the Organic CPU . It houses the `BioState`, `EcoMetrics`, and `OrganicCpuPolicy` structs, which represent the real-time telemetry and safety envelopes of the system . Functions like `read_biostate()` allow clients to query the current bio-scale metrics that inform the `OrganicCpuPolicy`'s decision-making loop, which can trigger actions like Allow, Degrade, or Pause based on the measured `BioState` . Like `organiccpualn`, this crate ensures that interactions with the system's core operational state are channeled through a controlled, typed interface, preventing ad-hoc or unauthorized computations that could lead to inconsistent or unsafe conclusions .

However, the most powerful and consequential component of the anchor-trio is `sovereigntycore`. This crate is the sole portal for submitting and evaluating `EvolutionProposalRecord`s, representing the only way a client can request a state mutation . When a client constructs a proposal, it does not write to the target shard directly. Instead, it uses bindings to call a function within `sovereigntycore`, such as `evaluate_proposal(...)`, passing the serialized proposal for processing . Inside `sovereigntycore`, the proposal is subjected to the full, ordered guard pipeline: RoH, neurorights, stake/token, and viability checks . The verdict—Allowed, Rejected, or Deferred—is then returned to the client. Crucially, only the "Allowed" verdict triggers the actual mutation of the target shards and the recording of the decision in the `.donutloop.aln` ledger . By centralizing this entire process, `sovereigntycore` acts as the final arbiter of change, ensuring that every potential evolution is rigorously vetted before it can take effect. This design makes it impossible for a client to bypass the guard pipeline, even if it possesses elevated privileges, because there is no alternative path to state mutation.

The following table summarizes the responsibilities and functions of each anchor-trio crate, illustrating how they collectively form the complete, secure API surface for interacting with the NeuroPC system.

Crate Name	Primary Responsibility	Key Exposed Functions/APIs	Role in Security Model
organiccpualn	Provides typed bindings for reading sovereign shards.	`load_rohmodel()`, `load_stake()`, `load_neurorights()`, etc.	Prevents direct file access; enforces schema compliance for all reads.
organiccpucore	Exposes core bio-cognitive state and policy.	`read_biostate()`, `get_ecometrics()`, `check_policy_envelope()`	Channels access to real-time telemetry through a controlled, typed interface.
sovereigntycore	Executes the full guard pipeline and evaluates proposals.	`evaluate_proposal(evolution_proposal_record)`	Acts as the single gatekeeper for all state mutations; rejects proposals that fail guard checks.

This unified API model is what allows NeuroPC to achieve Jupyter compatibility without compromising its security posture. The Jupyter notebook environment is not given special permissions or a direct connection to the system's internals. Instead, it runs a sandboxed client library that imports and uses the same anchor-trio crates as any other client . The behavior of this client library is entirely dictated by the `bostrom-sovereign-kernel-v2.ndjson` manifest file, which tells it where to find the shards and which guard configuration to use . This manifest acts as a portable constitution, allowing the same client code to operate securely across different deployments or experimental configurations simply by pointing to a different manifest. Nothing inside the notebook itself is privileged; it is merely an automagic front-end speaking the protocol defined by the anchor-trio crates and the sovereign manifest . This design elegantly solves the problem of providing a powerful, interactive research environment while maintaining the strict, rule-based integrity of the underlying sovereign system.

Jupyter Integration as a Client-Side Safety Boundary

The integration of Jupyter notebooks into the NeuroPC ecosystem is a critical test of the single-plane sovereignty model's robustness. Rather than viewing Jupyter as a privileged runtime requiring special treatment, the architecture frames it as just another client, bound by the same stringent safety and access-control rules that apply to all other components . This design choice is fundamental to preserving the system's integrity. The goal is to provide researchers with a powerful tool for analysis and experimentation without granting them a backdoor to manipulate the system's core state. This is achieved by establishing a clear and unbreakable boundary between the notebook's environment and the sovereign data plane, mediated by the anchor-trio crates and strictly governed by

the `bostrom-sovereign-kernel-v2.ndjson` manifest . This section examines how Jupyter's role is redefined from a tool of direct control to a sophisticated proposal authoring surface, reinforcing the safety boundaries of the system.

The core principle of Jupyter integration is that the notebook environment should never gain privileged write access to the canonical shards like `.rohmodel.aln`, `.stake.aln`, or `.lifeforce.aln` . Instead, the notebook acts as a rich client for analysis and drafting. Within the notebook, users can leverage its strengths for scientific computing: aggregating data from various sources like `.biosession.aln`, QPU output shards, and logs to create insightful plots and visualizations; inspecting the history of the donutloop ledger to understand past evolution decisions; and tracking RoH trends over time . This analytical capability is essential for research and debugging. However, the crucial distinction lies in the fact that all this activity is fundamentally read-only in terms of impacting the live system state .

The notebook's true power is unlocked when it moves from analysis to proposal authoring. Researchers can use the notebook to prototype hypothetical changes safely in memory . For example, they can simulate the effect of tightening an envelope or adjusting the weights in the RoH model to see how the calculated RoH value would respond under different scenarios . This in-memory prototyping is vital for exploring the consequences of a change before formally submitting it. However, this simulation exists entirely within the notebook's process and does not touch the real system's state. The moment a tweak seems beneficial and is ready to be enacted, the workflow shifts from simulation to official actuation. The notebook code serializes a candidate `EvolutionProposalRecord` that conforms to the canonical schema of the `.evolve.jsonl` shard and passes it off to a Rust binding for evaluation by the `sovereigntycore` crate . This act transforms the notebook from a tool of direct control into a highly effective drafting interface for the true decision-maker: the guard pipeline.

This functional split is the key to maintaining safety. Inside the notebook, work is confined to "analysis & visualization + proposal drafting" . The notebook is an expert system for thinking and drafting. The `sovereigntycore` plus the shard schemas are the expert systems for deciding and actuating. This division of labor preserves the integrity of the single data/guardrail plane even during interactive, exploratory sessions . The notebook facilitates the "what if" questions, but the guard pipeline provides the definitive "yes" or "no." This ensures that the final decision to mutate the system's state is always made by the trusted, centralized authority (`sovereigntycore`) and never by the whims or potential errors of an interactive session. The notebook becomes a "just another shell" that speaks the same protocol as any other client, parameterized by the same sovereign manifest and using the same anchor-trio crates .

The `bostrom-sovereign-kernel-v2.ndjson` manifest file is the linchpin that wires the Jupyter client to the correct system configuration and security policy . When the Jupyter kernel starts, its client library reads this manifest to discover the precise locations of all canonical shards, the versions of the schemas to use for deserialization, and, most importantly, the exact ordered sequence of the guard pipeline to enforce . This manifest serves as the single source of truth, configuring the client's behavior entirely from external, declarative data rather than hardcoded values. This makes the system highly configurable and portable. A single Jupyter client library can be deployed across different NeuroPC instances or experimental setups simply by providing it with the appropriate manifest file. This reinforces the idea that the system's "constitution" is self-describing and that all participants, from the kernel to the end-user's notebook, derive their operational rules from this central document. The manifest ensures that the notebook client is always aligned with the current state of the sovereign data plane and its associated guardrails .

In essence, the Jupyter integration pattern is a masterclass in designing for safety through constraint. By treating the notebook as a standard, unprivileged client and forcing all actuation through the `sovereigntycore` gate, the architecture avoids the common pitfalls of interactive AI systems, where user-friendly interfaces often introduce shortcuts that undermine safety features. The notebook provides the freedom to explore and innovate, but it does so within a rigidly defined framework that channels all innovations through a rigorous, verifiable, and auditable process. This ensures that while the system can evolve and adapt, it does so in a manner that is consistent with its foundational principles, preserving its sovereignty and integrity.

The Evolution Lifecycle and Donutloop Ledger as the Proving Ground

The theoretical elegance of NeuroPC's guard pipeline and embedded verification mechanisms is ultimately validated through their practical application in the real-world operational dynamics of the system. The evolution proposal lifecycle and the donutloop ledger serve as the crucible—the proving ground—where every safety guarantee is tested. This lifecycle is the tangible manifestation of the single-plane sovereignty model in action, demonstrating how abstract principles are translated into verifiable events. The journey of an `EvolutionProposalRecord` from a tentative draft in a researcher's notebook to a confirmed, immutable entry in the system's history is the primary evidence of the system's integrity. This section dissects this lifecycle and the donutloop's role as a

cryptographic audit trail, showing how they provide empirical proof that the enforcement logic has been correctly and consistently applied.

The evolution lifecycle begins when a client, such as a Jupyter notebook, decides to propose a change. This initial phase is characterized by analysis and in-memory prototyping . The researcher aggregates data from various shards like `.biosession.aln` and `.ocpulog` to understand the current state, plots RoH trends, and experiments with hypothetical adjustments to system parameters without affecting the live system . This is the "thinking and drafting" stage. When a potential improvement is identified, the notebook code serializes this candidate change into a well-formed `EvolutionProposalRecord`. This record is a structured data object that specifies the type of change, its scope (e.g., day-to-day tuning, archchange, lifeforcealteration), the expected RoH delta, and any required token usage . This serialization must conform precisely to the schema of the `.evolve.jsonl` shard, acting as the first formal checkpoint .

Once serialized, the proposal is submitted to `sovereigntycore` for evaluation . This marks the start of the core enforcement phase. `sovereigntycore` reads the `bostrom-sovereign-kernel-v2.ndjson` manifest to determine which shards to load and which guard pipeline to execute . It then materializes the relevant shard data into typed Rust structs and proceeds to run the ordered sequence of checks. The proposal is first vetted by the RoH guard, which calculates the impact of the proposed change on the Risk of Harm metric. If the resulting RoH would exceed the constitutional limit of 0.3, the proposal is immediately rejected . If it passes, it moves to the neurorights guard, which verifies compliance with legal and ethical policies . Next, the stake/token guard validates the proposer's identity and permissions . Finally, the viability check, using the Tsafe/ viability kernel geometry from `.vkernel.aln`, ensures the proposed state remains within the mathematically safe set . This multi-stage evaluation is the heart of the system's safety mechanism.

The outcome of this evaluation is the pivotal moment of the lifecycle. There are three possible verdicts: Allowed, Rejected, or Deferred . A "Rejected" verdict means the proposal violated one of the guards and is discarded. A "Deferred" verdict might indicate a temporary condition, such as a lack of available tokens or a need for further human review. In both cases, the reason for the decision is recorded in the donutloop ledger, providing transparency . The "Allowed" verdict is the rarest and most significant outcome. Upon receiving this, `sovereigntycore` performs the actual state mutations, updating the target shards (e.g., writing new weights to `.rohmodel.aln`). Immediately following this, it atomically appends a new entry to the `.donutloop.aln` ledger . This atomicity is crucial; it guarantees that the state change and the audit record are treated as a single,

indivisible transaction. If one part fails, the entire operation is rolled back, preserving the system's consistency.

It is at this final step that the donutloop ledger reveals its true purpose. Described as a "hash-chained audit log," the `.donutloop.aln` shard is more than a simple history book; it is a tamper-evident cryptographic ledger . Each new entry contains the hash of the previous entry, creating a chain of custody for the system's evolution. This structure makes it computationally infeasible to alter past entries without breaking the chain and being detected. The ledger contains a complete, cryptographically signed record of every `EvolutionProposalRecord` processed by the system, along with the final decision (Allowed/Rejected/Deferred) and the context of the evaluation . This donutloop becomes the ultimate, verifiable source of truth for what changes were *actually* made to the system's state. An auditor or another system participant can replay the entire evolution process by starting from the genesis entry and walking through the chain, verifying at each step that the guard pipeline was correctly applied and that the state transitions were lawful.

The donutloop ledger thus serves as the public proof that the `sovereigntycore` kernel is faithfully executing its duties. It provides an immutable, append-only record that can be independently verified, demonstrating the system's adherence to its own constitution. This turns the abstract concept of "sovereignty" into a concrete, observable property. The integrity of the donutloop depends on the immutability of the shard it resides in and the cryptographic linking of its entries. Any attempt to manually edit a shard or forge an entry would break the hash chain, leaving a clear and undeniable trace of the tampering. Therefore, the evolution lifecycle and the donutloop ledger together form a powerful feedback loop: the lifecycle is the process of enforcement, and the donutloop is the permanent, verifiable proof of that enforcement. This combination is what gives the single-plane sovereignty model its teeth, providing a robust mechanism for ensuring accountability and trustworthiness in a complex, evolving system.

Synthesis of the Single-Plane Sovereignty Model

The NeuroPC architecture, as detailed through its sovereign guard pipeline, embedded verification mechanisms, and client-side safety boundaries, presents a cohesive and theoretically robust model for managing a complex adaptive system. At its core, the design is built upon the "single-plane sovereignty model," a paradigm that collapses the traditional distinctions between the data plane, control plane, and execution plane into a

unified, self-verifying system [20] [23]. In this model, all real state and policies reside within a single, canonical data plane composed of ALN/JSON/NDJSON shards . The enforcement logic is not a separate, privileged component but is tightly coupled with the data it governs, creating a closed loop of rule definition, validation, and actuation. This synthesis integrates the key findings from the preceding sections to articulate the holistic nature of this architectural philosophy.

The foundation of this model is the **Constitutional NDJSON**, a single, machine-readable document (`bostrom-sovereign-kernel-v2.ndjson`) that serves as the system's portable constitution . This manifest declares the location and schema of every critical shard, defines the cross-file invariants, and, most importantly, specifies the exact, ordered sequence of the guard pipeline . By externalizing this core policy to a declarative file, the system's rules become explicit, inspectable, and modifiable through the same evolutionary process they are meant to govern. This manifest acts as the source of truth, configuring both the system's kernel (`sovereigntycore`) and all its clients, including Jupyter notebooks, ensuring a uniform and consistent interpretation of the law across the entire ecosystem .

The enforcement of this constitution is carried out by the **Guard Pipeline**, a multi-stage, sequential check orchestrated by the `sovereigntycore` crate . The strict ordering—RoH guard, neurorights guard, stake/token guard, followed by donutloop logging—implements a clear hierarchy of priorities, ensuring that fundamental safety and ethical constraints are always checked first . This pipeline is the embodiment of the system's will, a deterministic engine that translates a proposed change into a definitive verdict. The `sovereigntycore` crate acts as the sole gatekeeper, the kernel of authority, preventing any direct manipulation of the sovereign shards and channeling all state mutations through this rigorous evaluation process . This centralized control is the primary mechanism for upholding the system's integrity.

Complementing the runtime enforcement of the guard pipeline are the **Embedded Verification Mechanisms** that make the data plane itself a site of verification. The system leverages a multi-layered stack of checks: 1. **Formal Shards Schemas**: Each shard type comes with a formal schema that acts as a static contract, defining its structure and constraints. Typed Rust bindings generated from these schemas provide compile-time guarantees of data integrity [1]. 2. **CI-Enforced Invariants**: A pre-submission layer of verification, performed by Continuous Integration pipelines, catches errors and ensures proposed changes adhere to structural and coding standards before they ever reach the live evolution stream . 3. **Tsafe/Viability Proofs**: The most advanced mechanism is the inclusion of mathematical proofs within the `.vkernel.aln` shard. These proofs provide formal, non-heuristic guarantees that the system's dynamics will remain stable and

bounded, moving beyond heuristic thresholds to encompass the global safety of the system's state space [16].

Together, these layers transform the data plane from inert storage into an active, verifiable artifact of the system's state and principles. The data itself becomes the evidence of its own validity.

Finally, the entire architecture is designed to treat **Everything Else as a Client**. Interactive environments like Jupyter are not given special privileges but are instead sandboxed as unprivileged clients that communicate with the sovereign kernel through the restricted API surface provided by the anchor-trio Rust crates . This forces a clear separation between analysis/drafting (which happens in the notebook) and actuation (which is handled by the guard pipeline). The donutloop ledger serves as the immutable, cryptographic audit trail of this entire process, providing an external, verifiable proof that the `sovereigntycore` has been faithfully executing its duties .

In conclusion, NeuroPC's architecture is a sophisticated implementation of a self-sovereign system. It achieves sovereignty not through isolation, but through a combination of rigorous rule-based enforcement, deep mathematical guarantees, and transparent, auditable processes. The single-plane model succeeds by making the system's laws (the manifest and schemas), its enforcers (the guard pipeline), and its proof of compliance (the donutloop ledger) co-located and interdependent. While the provided materials establish the high-level design and principles, practical considerations such as the performance overhead of multi-stage proofs, the specifics of the viability proof generation, and the human-in-the-loop aspects of governance remain areas for further investigation. Nevertheless, the presented model offers a compelling blueprint for building AI systems that are not only intelligent but also verifiably safe, predictable, and accountable to their own foundational principles.

Reference

1. JSONSchemaBench: A Rigorous Benchmark of Structured ... https://arxiv.org/html/2501.10868v3
2. Foundations of JSON Schema | Request PDF https://www.researchgate.net/publication/312633891_Foundations_of_JSON_Schema

3. learning to generate structured output with schema ... https://arxiv.org/pdf/2502.18878
4. (PDF) Brain-MCP: A Fully Homomorphic Neuro-Symbolic ... https://www.researchgate.net/publication/393129669_Brain-MCP_A_Fully_Homomorphic_Neuro-Symbolic_Protocol_for_Governing_System_3_Cognition_and_Restoring_Semantic_Scarcity
5. Learning to Generate Structured Output with Schema ... https://arxiv.org/html/2502.18878v1
6. Data model, Query languages and Schema specification https://www.researchgate.net/publication/316849641_JSON_Data_model_Query_languages_and_Schema_specification
7. Security and Cyber Resilience with Power11 https://www.redbooks.ibm.com/redpieces/pdfs/sg248595.pdf
8. Cisco Compute Intersight Hardening Guide White Paper https://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/intersight/compute-intersight-hardening-guide-wp.html
9. Dylan Seals posted on the topic https://www.linkedin.com/posts/dylanthomasseals_the-current-aerospace-paradig-championed-activity-7419824898912460801-jcE-
10. A Review on Blockchain Technology, Current Challenges ... https://dl.acm.org/doi/full/10.1145/3700641
11. (PDF) Analysis of Workloads for Cloud Services https://www.academia.edu/96340801/Analysis_of_Workloads_for_Cloud_Services
12. Foundations of JSON Schema | Proceedings of the 25th ... https://dl.acm.org/doi/10.1145/2872427.2883029
13. JSON Structure: A JSON schema language you'll love https://techcommunity.microsoft.com/blog/messagingonazureblog/json-structure-a-json-schema-language-youll-love/4476852
14. Artificial Intelligence Jul 2025 https://www.arxiv.org/list/cs.AI/2025-07?skip=300&show=2000
15. Artificial Intelligence https://arxiv.org/list/cs.AI/new
16. Hints and Principles for Computer System Design 1. Introduction https://web3.arxiv.org/pdf/2011.02455v2
17. Autonomous Agents on Blockchains: Standards, Execution ... https://www.arxiv.org/pdf/2601.04583
18. Computer Science https://arxiv.org/list/cs/new

19. Computer Science https://www.arxiv.org/list/cs/new?skip=150&show=500
20. Hints and Principles for Computer System Design https://arxiv.org/pdf/2011.02455
21. Computer Science https://www.arxiv.org/list/cs/new?skip=25&show=1000
22. Computer Science Apr 2025 https://www.arxiv.org/list/cs/2025-04?skip=7000&show=2000
23. Computer Science https://www.arxiv.org/list/cs/new?skip=275&show=2000
24. Computer Science https://www.arxiv.org/list/cs/recent?skip=1318&show=2000