The Tree-of-Life Framework: An Architectural Blueprint for a Non-Actuating, Configurable Observer in Sovereign Neuromorphic Systems

Architectural Blueprint: Defining the TreeofLife.rs and TreeofLife.md Artifacts

The successful integration of the Tree-of-Life as a non-actuating, educational component within the NewRow-Print!/NeuroPC sovereignty stack hinges on a meticulously defined architectural blueprint. This blueprint is composed of two primary artifacts: a Rust module, TreeofLife.rs, which contains the executable logic, and a Markdown specification, TreeofLife.md, which serves as the definitive source of truth for its design, purpose, and operational constraints . Together, these files establish a framework that prioritizes safety, modularity, and adherence to the core principle of non-actuation. The architecture is designed to function as a passive observer, reading system state without ever influencing it, thereby providing valuable diagnostic and educational insights without introducing any new pathways for control or actuation beyond those already sanctioned by the sovereign kernel.

The TreeofLife.rs artifact is conceived as a self-contained Rust module, intended to be placed at a location such as src/tree_of_life/TreeofLife.rs or within an appropriate existing crate like neuro_automagic_core/src/tree_of_life.rs . Its entire purpose is to serve as a computational view, transforming complex, low-level neuromorphic data into a more intuitive, high-level representation based on the 14 TREE assets . The core of its architecture is built around pure data structures and side-effect-free functions, which is fundamental to its observer role. The module defines several key structs, all equipped with Serde's Serialize and Deserialize traits, making them suitable for serialization into formats like JSON and NDJSON stackoverflow.com . These structs form a strict, one-way data pipeline.

The foundational struct is TreeOfLifeInput, which acts as the sole interface for feeding external data into the module . It is designed to be a simple container holding the necessary inputs for a single neuromorphic snapshot: the current capability_state (an enum representing the CapabilityState lattice), the roh_score (a scalar value representing the Risk-of-Harm score), a BiophysicalEnvelopeSnapshot containing physiological metrics, and optional indices for evolution and epoch counting . By encapsulating all inputs into a single struct, the API becomes clean, explicit, and easy to serialize, reinforcing the module's role as a passive consumer of data rather than an active participant in the system's state machine. This input struct ensures that the module has access to all the information it needs from the sovereign stack—envelopes, risk models, and capability levels—without requiring direct access to underlying hardware drivers or state-mutation functions .

From this input, the main computational function, pub fn from_snapshot(input: &TreeOfLifeInput) → TreeOfLifeView, constructs the central output struct, TreeOfLifeView . This struct is a flat collection of 15 scalar fields, one for each TREE asset (blood, oxygen, wave, h2o, time, decay,

lifeforce, brain, smart, evolve, power, tech, fear, pain, nano), each typed as f32 and normalized to a [0.0, 1.0] range . The normalization process, performed by a helper function clamp01, ensures that all outputs are bounded and dimensionless, facilitating comparison and visualization across different assets and timepoints . The immutability of these structs is paramount; from_snapshot takes an immutable reference to the input and returns a newly constructed TreeOfLifeView. There are no methods that mutate the TreeOfLifeView itself or any other shared state. This functional purity guarantees that calling the function multiple times with the same input will always yield the same output, a critical property for a diagnostic tool whose results must be reproducible and reliable
arxiv.org
.
To complete the observable view, a second output struct, TreeOfLifeDiagnostics, is provided . This struct contains advisory information derived solely from the TreeOfLifeView. It includes a vector of human-readable labels (e.g., "balanced", "overloaded") and boolean flags like cooldown_advised . The function pub fn diagnostics(view: &TreeOfLifeView) → TreeOfLifeDiagnostics computes these diagnostics based on predefined logical conditions applied to the asset scores . For instance, the "overloaded" label is generated if fear or pain exceed a certain threshold, and cooldown_advised is true if any of several stress indicators are high . Like from_snapshot, this function is also pure and side-effect-free, operating only on the provided view data. The existence of a separate diagnostics struct cleanly separates the raw asset computation from the higher-level interpretation and labeling, promoting modularity and clarity in the codebase.
The entire module is wrapped within a public empty struct, pub struct TreeOfLife, which contains only impl blocks for its functions . This pattern is a deliberate architectural choice to signal that the module is a namespace for a set of utility functions rather than a stateful object that can be instantiated. It reinforces the idea that there is no internal state to manage or mutate, further cementing its identity as a read-only diagnostic surface. This structure is explicitly documented in the accompanying TreeofLife.md specification, which codifies the module's invariants, particularly the absolute prohibition against actuation . The specification states unequivocally that TreeofLife.rs MUST NOT call hardware drivers, modify CapabilityState, consent tokens, or alter any of the core ALN files (.stake.aln, .rohmodel.aln, etc.) except through standard logging mechanisms . This invariant is the most critical aspect of the architecture, ensuring that the Tree-of-Life layer remains subordinate to the sovereignty core and never becomes a backdoor for unauthorized capability changes.
The TreeofLife.md file serves as the authoritative documentation for the entire system . Its placement, suggested to be in a directory like docs/tree_of_life/ or specs/, alongside the Rust module, ensures that developers, auditors, and users have a single, accessible source for understanding its purpose and behavior . The document begins by stating the module's purpose: an educational, fairness-oriented explanation surface for AI-chat, HUDs, and audits, with zero authority over capability or consent . It then formally lists the actions the module MUST NOT perform and MAY perform, creating a clear boundary for its operation . This part of the specification directly translates the user's high-level requirement for a non-actuating layer into concrete, enforceable rules.
The specification then details the inputs and outputs, mirroring the structs defined in the Rust code but in prose . It elaborates on the contents of TreeOfLifeInput, specifying the types and sources of each field, such as BiophysicalEnvelopeSpec shards for cognitive load and sleep

arousal . For TreeOfLifeView, it provides a detailed description of each of the 14 assets, explaining their conceptual meaning and their relationship to the underlying neuromorphic signals . For example, it clarifies that BLOOD is derived from the heart rate envelope and WAVE from EEG bandpowers, grounding the abstract assets in concrete, existing measurements . This level of detail in the specification is crucial for ensuring that the implementation in TreeofLife.rs is correct and that future developers understand the rationale behind each mapping. The specification also includes a concrete JSON example of a logged snapshot, showing how the TreeOfLifeView and diagnostics fit into the broader context of a neuromorphic record, including the parent capability_state and roh_score . This practical example serves as both documentation and a test case for serialization compatibility.

Finally, the specification outlines the governance and invariants that govern the module's use . It reiterates the non-actuation rule, the principle of being a view over the existing safety stack, and the requirement for provenance on all numeric mappings . The concept of "provenance" is particularly important; it mandates that any numerical values or scaling factors used in the mappings (e.g., the 0.3 ceiling for RoH) must be explicitly documented and correspond to values already established within the NewRow-Print!/NeuroPC governance stack, such as those found in .rohmodel.aln or ALN shards . This prevents the introduction of arbitrary or unvalidated constants into the system. The specification also touches upon the concept of fairness, noting that while the module can compute metrics, the final determination of fairness and any associated actions must still pass through the existing policy engine . This correctly positions the Tree-of-Life as a source of information and potential triggers for human review or higher-level automated processes, but never as an arbiter of policy itself. Together, TreeofLife.rs and TreeofLife.md form a complete and robust architectural blueprint, defining a clear, safe, and extensible framework for integrating the Tree-of-Life observer layer into the sovereignty stack.

Artifact

Path

Role

Key Components

Rust Module

src/tree_of_life/TreeofLife.rs

Execution: Implements the computational logic to map neuromorphic snapshots into the 14 TREE assets. It is a pure, side-effect-free utility library.

TreeOfLifeInput (input struct)

TreeOfLifeView (asset struct)

TreeOfLifeDiagnostics (diagnostic struct)

TreeOfLife (namespace struct with pure impl functions)

Specification

docs/tree_of_life/TreeofLife.md

Documentation & Governance: Serves as the official specification, defining the module's purpose, asset mappings, invariants, and governance rules. It is the source of truth for developers and auditors.

Purpose and Non-Actuation Invariants

Inputs (TreeOfLifeInput) and Outputs (TreeOfLifeView, TreeOfLifeDiagnostics)

Asset-by-Asset Computation Logic

Example JSON Snapshot

Governance and Provenance Rules

This dual-artifact approach creates a powerful feedback loop between code and specification. The code implements the logic described in the spec, and the spec documents the behavior of the code, ensuring consistency and maintainability. It is a mature software engineering practice that is essential for building trust in a safety-critical system like a sovereign neuromorphic architecture. The architecture is intentionally minimal and focused, avoiding the creation of complex state machines or observers that could introduce unforeseen interactions. Instead, it provides a simple, reliable function that transforms data, leaving all decision-making and actuation to the sovereign core it observes.

Serialization and Integration: Ensuring Compatibility with Canonical Data Streams

A cornerstone of the Tree-of-Life framework is its seamless integration with the existing data infrastructure of the NewRow-Print! system, primarily through serialization compatibility with canonical data streams like .evolve.jsonl and .donutloop.aln . This design choice is not merely a technical convenience but a strategic decision rooted in the principles of auditability, modularity, and sovereignty. By treating its outputs as just another type of loggable data, the Tree-of-Life module avoids creating proprietary or isolated data silos. Instead, it contributes to the official, pre-validated shell surfaces that every controller and analysis tool within the ecosystem is already designed to respect and process . This ensures that the educational and diagnostic views provided by the Tree-of-Life are intrinsically linked to the official timeline of system events and decisions, making them a first-class citizen in the system's audit trail.

The foundation for this compatibility is laid directly in the architectural blueprint of the TreeofLife.rs module. The TreeOfLifeView and TreeOfLifeDiagnostics structs are defined using standard Rust primitives (like f32 and String) and are annotated with the #[derive(Debug, Clone, Serialize, Deserialize)] macro from the Serde library . Serde is the de facto standard for serialization in the Rust ecosystem, and its adoption here is critical stackoverflow.com . The Serialize trait allows instances of these structs to be converted into various data formats, with JSON being the most common and relevant for this application. JSON is a lightweight, text-based, language-independent data interchange format that is easy for humans to read and write and easy for machines to parse and generate www.w3.org +1 . This makes it an ideal choice for logging and subsequent analysis. The ability to serialize to JSON means that a TreeOfLifeView instance can be easily embedded within a larger JSON object that represents a full neuromorphic snapshot, as demonstrated in the example provided in TreeofLife.md .

The integration strategy prioritizes logging over real-time, direct API calls from front-end applications. While the Rust module exposes a pure function, from_snapshot, that can be called by any part of the Rust codebase, the primary consumption pattern envisioned for external systems like an Unreal Engine HUD or an AI-chat explanation layer is to consume the serialized logs . This follows a decoupled architecture where the producer (the sovereignty core computing the Tree-of-Life view) and the consumer (the visualization or explanation engine) do not need to be tightly coupled. The producer simply writes a standardized log entry to a canonical stream, and the consumer reads from that same stream. This pattern has several significant advantages. First, it ensures that every view rendered by a frontend is based on an officially recorded event, preventing discrepancies that could arise from asynchronous updates or direct inter-process communication. Second, it creates a permanent, immutable record of the

system's state from the Tree-of-Life perspective, which is invaluable for post-mortem analysis, debugging, and long-term auditing
www.arxiv.org
. Third, it simplifies the frontend implementations. An Unreal HUD or an AI-agent doesn't need to know about the intricacies of the Rust module's API; it only needs to know how to parse a standardized JSON log file, a task for which numerous libraries exist in virtually every programming language.
The canonical data streams themselves provide the logical destination for these logs. The .donutloop.aln file is described as a ledger of decisions, likely containing records of capability transitions and major system events . Logging a TreeOfLifeView snapshot here would associate the biophysical state with a specific point in the system's capability history. The .evolve.jsonl file, on the other hand, is a line-delimited JSON stream for per-event records, tracking the evolution of the system over time . This is the perfect place to log a TreeOfLifeView for every neuromorphic cycle or epoch, creating a high-resolution time-series of the system's biophysical state. The user's clarification explicitly prioritizes this logging-first approach, emphasizing that the Tree-of-Life should serialize cleanly into these existing streams because they are the pre-validated surfaces that all controllers must respect . This aligns perfectly with best practices in secure system design, where data flows are channeled through well-defined, monitored, and governed pipes.
The example JSON snippet provided in the TreeofLife.md specification illustrates exactly how this integration would work in practice . It shows a top-level JSON object containing the capability_state, roh_score, and then nested objects for tree_of_life_view and diagnostics . This structure is highly readable and analyzable. Tools designed to parse .donutloop.aln or .evolve.jsonl would not require any special handling for the Tree-of-Life data; it would simply appear as additional, structured fields within each log record. This composability is a key feature of the JSON format and is a primary reason for its selection
docs.oracle.com
+1
. For example, a data analyst could use standard command-line tools like jq or scripting languages like Python to extract all lifeforce values over time, correlate them with roh_score values, and plot their relationship without needing to write custom parsers for the Tree-of-Life data itself
www.kaggle.com
.
Furthermore, this logging-centric approach facilitates schema evolution, a critical consideration for a long-term project. As the Tree-of-Life specification evolves—for instance, if a new asset is added or the computation of an existing asset is refined—the log format can change accordingly. While breaking changes to a canonical stream like .evolve.jsonl would be disruptive, the modular nature of JSON makes it easier to manage non-breaking changes. For example, adding an optional field to the TreeOfLifeView JSON object is a non-breaking change that older parsers can safely ignore
py.iceberg.apache.org
. If a breaking change were ever necessary, it could be managed through a versioning scheme in the log file's metadata, allowing consumers to adapt their parsing logic. This contrasts with a tight API coupling, where a change in the Rust function signature would require corresponding changes in every single client application.

In the context of Unreal Engine visualization, this model is equally effective. An Unreal plugin could be developed to monitor the .evolve.jsonl file for new entries. Upon detecting a new line, it would parse the JSON, extract the TreeOfLifeView object, and update the corresponding HUD elements (e.g., gauges for BLOOD and OXYGEN, a text label for diagnostics.labels). This decouples the visualization logic from the core neuromorphic processing loop, improving performance and stability. The Unreal application would be a passive reader of the official state, not an active participant in its calculation. Similarly, for an AI-chat explanation agent, the agent could be provided with the contents of a recent log file. It could then analyze the sequence of TreeOfLifeView snapshots to explain why a particular decision was made or to summarize the system's overall state in natural language. The structured, serialized data provides a solid foundation for such an AI-driven explanation layer
link.springer.com
.
While the primary interaction pattern is through log files, the pure-Rust API is not redundant. It remains a vital tool for internal system components and unit testing. For example, a component responsible for writing the logs could call TreeOfLife::from_snapshot internally. Additionally, developers can use this API in tests to verify the correctness of the asset mappings in isolation, without having to simulate the entire neuromorphic loop and log-writing process. However, the framework's design philosophy makes it clear that this internal API is an implementation detail of the logging mechanism itself, not a public-facing service for general-purpose querying. The official, documented, and auditable channel for accessing Tree-of-Life data is always the canonical log stream. This design elegantly balances the need for a simple, decoupled integration model for external consumers with the need for a clean, testable internal API for the system's own components.
Configurable Fairness Diagnostics: Advisory Labels and Governance
A critical challenge in designing the Tree-of-Life observer layer is the implementation of its fairness diagnostics. The user's requirements demand that these diagnostics—such as labels for 'overloaded' or 'cooldown-advised' states—be treated as configurable, advisory parameters rather than as hard policy rules . This distinction is paramount to preserving the sovereignty of the core neuromorphic stack. The Tree-of-Life must provide insightful, potentially actionable signals, but the authority to act on those signals must reside exclusively within the pre-existing policy invariants like RoH ≤ 0.3, neurorights constraints, and the OwnerDecision gate . The research framework must therefore define a robust mechanism for configuring these diagnostic thresholds and a clear governance model that enforces their advisory-only status.
The initial implementation of the TreeOfLife module in TreeofLife.rs demonstrates a proof-of-concept for this functionality. The diagnostics function computes labels and flags based on hardcoded constant thresholds, such as checking if view.fear > 0.6 or view.pain > 0.6 to generate an "overloaded" label . While this is a valid starting point for development and testing, it violates the principle of configurability. Hardcoded logic is brittle and requires code modification and redeployment to change behavior, which is antithetical to a governance model that favors auditable, declarative policy tweaks. The user's clarification correctly identifies this gap and proposes a superior solution: moving these thresholds into an external configuration file that can be loaded by the sovereignty core at runtime .
The recommended framework involves three key steps. First, the diagnostics function should be refactored to accept a Config struct as an additional parameter. This Config struct would hold all the tunable parameters for the diagnostic logic, such as overload_threshold_fear: f32,

overload_threshold_pain: f32, decay_threshold: f32, etc. The function would then use these values from the config instead of the hardcoded constants. This immediately makes the diagnostic logic flexible and parameterized. Second, a default configuration struct should be provided within the module, initialized with reasonable, empirically-derived or expert-defined threshold values. This ensures the module is usable "out of the box" without any external configuration. Third, and most importantly, the sovereignty core (or a dedicated management service) should be responsible for loading a user-provided or administrator-defined configuration shard on startup. This shard could be a simple JSON file (e.g., treeoflife-config.json) or an ALN shard (e.g., treeoflife-config.aln), depending on the project's conventions for policy files . This shard would override the defaults from the Config struct before it is passed to the diagnostics function.

This approach offers several distinct advantages. From a governance perspective, it places the control over diagnostic sensitivity squarely within the established policy management channels. Changes to what constitutes an "overloaded" state are no longer a matter of modifying source code but a formal policy decision that is written to an auditable file, just like changes to .rohmodel.aln or .stake.aln . This makes the system's behavior more transparent and subject to review. From a practical standpoint, it allows operators to tune the sensitivity of the Tree-of-Life's warnings to match different contexts, users, or experimental goals without touching the core codebase. For example, a researcher in CapLabBench might want very sensitive alerts to detect subtle stress patterns, while a deployed system in CapControlledHuman might use more conservative thresholds to avoid nuisance alarms.

However, the framework must include a crucial piece of governance to prevent misuse. The TreeofLife.md specification must explicitly state that these diagnostic outputs are for advisory labeling only . It must reinforce that any logic that uses these labels to make a decision—such as transitioning a CapabilityState—must still pass through the full chain of sovereignty checks. The framework's design should block any direct wiring from a TreeOfLifeDiagnostics.cooldown_advised flag to a capability downgrade function. Such a connection would create a second, parallel rule system that could conflict with the primary RoH and neurorights invariants, undermining the security of the entire stack . The correct pattern is that the cooldown_advised flag could trigger a notification to an operator, who would then manually approve a transition, or it could be one of several inputs to a more complex, centrally-managed policy engine that also considers RoH, owner intent, and other factors before making a final decision. The specification should cite the existing ReversalConditions and OwnerDecision invariants as the ultimate arbiters of capability, ensuring the Tree-of-Life's voice is heard but not decisive .

The current implementation in TreeofLife.rs includes a placeholder field, fairness_imbalance: bool, in the TreeOfLifeDiagnostics struct, with a comment indicating it is intended to be computed at a higher level where multiple subjects or roles are visible . This correctly identifies the scope of the TreeOfLife module. The module itself, operating on a single snapshot, cannot meaningfully assess fairness across different users or roles. That is a systemic concern that requires correlating data from multiple sources over time. The framework should clarify this division of labor. The TreeOfLife module's responsibility is to provide the atomic, per-snapshot view of assets and stressors. Higher-level services or analytical pipelines would be responsible for collecting these views over time and across entities to compute aggregate fairness metrics. For example, a separate service could track the average drift of BLOOD vs. OXYGEN for all active sessions and report an imbalance if one asset is consistently depleted faster than others

under similar workload conditions. The fairness_imbalance flag in the TreeOfLifeDiagnostics struct would then be set by this higher-level service, not by the TreeOfLife module itself. This maintains a clean separation of concerns and prevents the observer module from becoming entangled in complex, cross-session analytics.
To implement this, the framework could define a clear protocol. When the sovereignty core processes a neuromorphic snapshot, it would:
Compute the TreeOfLifeInput.
Call TreeOfLife::from_snapshot to get the TreeOfLifeView.
Call TreeOfLife::diagnostics with the TreeOfLifeView and the loaded configuration to get the initial TreeOfLifeDiagnostics.
Pass the TreeOfLifeView (but not the diagnostics) to a separate FairnessAnalyzer service.
The FairnessAnalyzer would update its own state and, if a multi-session imbalance is detected, could publish a separate event or update a shared state that the UI or logging components could read to display a global fairness warning.
This multi-step process reinforces the advisory-only nature of the Tree-of-Life's immediate outputs. The core diagnostic logic, driven by configurable parameters, provides the raw material for both immediate warnings and long-term analytical insights. The governance model ensures that neither type of insight can bypass the sovereign decision-making hierarchy. The framework, therefore, successfully reconciles the need for sophisticated, adaptive diagnostics with the non-negotiable requirement for a secure, controlled, and auditable neuromorphic architecture.
Extensible Asset Hooks: Designing for Future Biophysical Proxies
A forward-looking design is essential for any system intended to evolve over time, especially one as complex as a sovereign neuromorphic architecture. The Tree-of-Life framework addresses this need through a deliberate design pattern for extensibility, providing explicit hooks for future biophysical axes like H2O (hydration) and NANO (event granularity) while rigorously binding runtime behavior to today's validated signals . This approach ensures that the module remains adaptable and can incorporate new scientific insights as they become available, without ever introducing speculative or fictional data paths that could compromise system integrity. The principle is to keep the present simple and grounded, while preparing the future with clear, documented pathways for expansion.
The current implementation of the TreeOfLife module in TreeofLife.rs and TreeofLife.md already embodies this philosophy for assets like H2O and NANO. The h2o field in the TreeOfLifeView struct is implemented as a neutral placeholder, currently fixed at a value of 0.5 . The accompanying specification explicitly states that this is a placeholder until hydration or slow-acting metabolic proxies are formally added as axes to the BiophysicalEnvelopeSpec . By keeping it neutral and not deriving it from any noisy or unreliable proxy, the implementation ensures that it has no "fictional signal path"; it does not influence the system's behavior in any way . Similarly, the nano asset is implemented as a simple proxy derived from the evolve_index, representing the granularity of logged changes in a purely informational capacity . These choices demonstrate a commitment to safety-first development: the module's behavior is strictly defined by what is known and validated today.

```
The framework's extensibility is formalized through two complementary mechanisms: structu
```

The TreeofLife.md specification plays a crucial role in documenting this extensibility. The framework should include a dedicated section, perhaps titled "Future Axes and Proxies," that

describes the protocol for adding new assets . This section would serve as the guide for developers wishing to expand the module's capabilities. It would outline the required steps: Formal Proposal: A new biophysical signal (e.g., a hydration proxy) must first be formally proposed and integrated into the main BiophysicalEnvelopeSpec and its associated ALN shards or datasets . This ensures the new signal undergoes the same rigorous validation and governance process as any other safety-critical data stream.

```
Implementation in TreeofLife.rs: Once the underlying signal exists, a developer can add t
```

Integration into from_snapshot: The main from_snapshot function would then be updated to conditionally call the new mapping function and populate the new field in the returned TreeOfLifeView struct if the underlying data is present .
This disciplined process ensures that the Tree-of-Life module never gets ahead of the rest of the system. It cannot start using a new signal until that signal is officially part of the system's trusted envelope of data. This prevents the module from becoming a vector for unvetted or unstable data to enter the system. Any new mapping logic would also need to be explicitly documented in TreeofLife.md, maintaining the transparency and traceability of the entire system .
This design pattern aligns with modern software development principles for building resilient and scalable systems. It promotes loose coupling and high cohesion. The TreeOfLife module is loosely coupled to the specific data sources it consumes; it only depends on the contract defined by the BiophysicalEnvelopeSnapshot struct. As long as that struct is updated to include new data, the TreeOfLife module can be extended without major refactoring. This is analogous to how database schemas evolve over time; new columns can be added, and applications can be updated to use them without disrupting the core functionality for existing data
py.iceberg.apache.org
+1
.
For example, let's say a new sensor for measuring skin conductance variability (SCV) becomes available and is integrated into the BiophysicalEnvelopeSpec. SCV could be a novel proxy for autonomic tone. According to the framework, a developer could propose adding an scv_norm field to the BiophysicalEnvelopeSnapshot. After governance approval, they could extend the TreeOfLife module by:
Adding pub scv_norm: Option<f32>, to TreeOfLifeView.
Writing a map_scv function that normalizes the SCV data.
Updating from_snapshot to populate scv_norm when the data is available.
All of this happens within a controlled, auditable process. The developer would not be able to simply start using the SCV data in a backdoor manner. The framework makes the extension process explicit, safe, and aligned with the overall governance of the NewRow-Print! project.
This approach also has implications for visualization and analysis. Frontend applications that consume the Tree-of-Life logs would need to be prepared to handle the optional fields gracefully. For instance, an Unreal HUD displaying the 14 assets would check if the hydration_norm field is Some(value) before attempting to render it. If it's None, the asset's gauge could be hidden or displayed in a disabled state. This robust handling of optional data is a direct benefit of the Option<T> design choice and ensures that the user interface remains stable even as the underlying data model expands. By combining structural flexibility in the code

with procedural rigor in the documentation, the Tree-of-Life framework provides a powerful and safe mechanism for continuous improvement and adaptation.

You already have all the ingredients for Tree-of-Life; what's missing is just the concrete, single-file artifacts wired into your existing sovereignty stack and envelope model. Below I'll stay tight and implementation-ready.

1. Where Tree-of-Life fits in NewRow-Print!

Within your current architecture, Tree-of-Life is a **read-only view** over three already-governed surfaces: CapabilityState (from `aln_core.rs`), RoH (from `.rohmodel.aln`), and BiophysicalEnvelope snapshots (from the `BiophysicalEnvelopeSpec` schema and shards).[1] [2]

- Capability states and transitions, consent, roles, and the composite `PolicyStack` are already defined as a pure Rust state machine (`CapabilityState`, `CapabilityTransitionRequest`, `Decision`). Tree-of-Life must never call or wrap these transition functions; it may only read the current state as a scalar enum value.[1]
- The biophysical space is already formalized as `BiophysicalEnvelopeSpec` / `EnvelopeAxis` with a layered minsafe/maxsafe vs minwarn/maxwarn model, capability binding, and RoH ceiling; the Tree-of-Life layer can safely consume a per-epoch "snapshot" of those features.[2]
- RoH is governed by `.rohmodel.aln` and enforced as a monotone score with a hard ceiling (0.30) in your capability engine; Tree-of-Life should treat RoH as an input scalar and never modify or re-weight it.[2] [1]

So the architectural placement you described—pure observer, no actuation, fed only by canonical, already-guarded values—is compatible with your existing Tier-2 envelope and Tier-1 policy engine design.[1] [2]

2. TreeOfLife.rs: single, pure, non-actuating module

A concrete, enforceable Rust layout that matches your narrative and your governance stack would look like this (only structure and invariants here, not full code):

Filename and path

- `src/tree_of_life/tree_of_life.rs` inside the sovereignty core workspace, or as `neuro_automagic_core/src/tree_of_life.rs` if you keep it in that crate.[1]

Core types (pure data only)

- `TreeOfLifeInput`
  - Fields (all read-only, `pub` for Serde, no methods with side effects):
    - `capability_state: CapabilityState` (reuse enum from `aln_core.rs`).[1]
    - `roh_score: f32` (current RoH, bounded elsewhere by `.rohmodel.aln` and guards).[2] [1]

        - `biophysical_snapshot: BiophysicalEnvelopeSnapshot` (a flattened, read-only struct mirroring your `BiophysicalEnvelopeSpec` axes—e.g., HRBPM, HRV RMSSD, EDATONICSCL, RESPRATE, EEG alpha CVE—at a single epoch).[2]

```
- Optional evolution indices: `evolve_index: Option<u64>`, `epoch_index: Option<u
```

    - `#[derive(Debug, Clone, Serialize, Deserialize)]` for JSON/NDJSON compatibility with `.evolve.jsonl`.[2] [1]

- `TreeOfLifeView`
    - Fifteen normalized scalars, each `f32` in $[0.0, 1.0]$: `blood`, `oxygen`, `wave`, `h2o`, `time`, `decay`, `lifeforce`, `brain`, `smart`, `evolve`, `power`, `tech`, `fear`, `pain`, `nano`.[2]
    - `#[derive(Debug, Clone, Serialize, Deserialize)]`.[2]
    - No mutation methods; the struct is constructed only via pure helper functions and `from_snapshot`.[2]
- `TreeOfLifeDiagnostics`
    - Fields such as:
        - `labels: Vec<String>` (e.g., `"balanced"`, `"overloaded"`, `"cooldown-window"`).
        - `cooldown_advised: bool`.
        - `fairness_imbalance: bool` (left as an externally set flag for multi-subject fairness logic, not computed inside this module).[1]
    - Also `#[derive(Debug, Clone, Serialize, Deserialize)]` and entirely pure (no internal state).[1]
- `TreeOfLifeConfig` (for advisory thresholds)
    - Holds tunable parameters for diagnostics only, e.g.:
        - `fear_overload_threshold: f32`, `pain_overload_threshold: f32`.
        - `cooldown_trigger_composite: f32` (e.g., condition on combined `decay`, `lifeforce`, `fear`, `pain`).
    - A `fn default()` that returns conservative defaults, with provenance documented in the spec and aligned with values from `.rohmodel.aln` / envelope shards where applicable.[1] [2]
    - This config is loaded and owned by the sovereignty core; Tree-of-Life only receives it as an argument.

Public functions

- `impl TreeOfLife` with only associated functions, no fields:
    - `pub fn from_snapshot(input: &TreeOfLifeInput) -> TreeOfLifeView`
        - Computes all 15 TREE asset scores as normalized scalars using internal, pure helpers such as `map_blood(&snapshot) -> f32`.[2]
        - Calls a local `fn clamp01(x: f32) -> f32` to guarantee each field is in `[0.0, 1.0]`.[2]

        - Never mutates external state, never uses `unsafe`, never calls any driver, network, file write, or capability transition function.[1] [2]
        - Deterministic: same input → same `TreeOfLifeView`. This matches your requirement for reproducible diagnostics.[2]
    - `pub fn diagnostics(view: &TreeOfLifeView, config: &TreeOfLifeConfig) -> TreeOfLifeDiagnostics`
        - Applies simple, documented rules, e.g.:
            - Add `"overloaded"` to `labels` if `view.fear > config.fear_overload_threshold` or `view.pain > config.pain_overload_threshold`.
            - Set `cooldown_advised` if any combination of `decay`, `lifeforce`, `fear`, `pain` crosses configured advisory limits.
        - Pure and side-effect-free; it only returns a new diagnostics struct.[1]
    - No methods to write logs, mutate capability, or access `.stake.aln`, `.rohmodel.aln`, `.donutloop.aln`; those are handled by existing kernel components and log writers. Tree-of-Life's outputs are just serialized and injected into `.evolve.jsonl` / `.donutloop.aln` by the logging layer you already have.[1]

Non-actuation enforcement

You already rely on ALN and kernel-level guards like `CapabilityGuard` and RoH enforcement; Tree-of-Life must be documented as:

- "Observer-only": it may be called by code that is itself gated by `.neuro-cap.aln` and `.stake.aln`, but it never initiates a transition or writes policy files.[1]
- "Never-actuate": the module is forbidden to call any code paths that can affect devices, envelopes, capability, or consent; this is captured in TreeofLife.md invariants and can be enforced by code reviews and lints (e.g., forbidding particular imports).[2] [1]

3. TreeofLife.md: authoritative, governance-aligned spec

A concrete placement and content structure for the spec that ties into your existing documentation style:

Path

- `docs/tree_of_life/TreeofLife.md` or `specs/tree_of_life/TreeofLife.md`, side-by-side with the Rust module and your existing ALN/biophysical spec documents.[2] [1]

Suggested sections (all non-fictional, grounded in current artifacts)

1. **Purpose and scope**
    - Declare Tree-of-Life as an "educational, fairness-oriented explanation surface for AI chat, HUDs, and audits with zero authority over capability or consent."[1]
    - Explicitly state: "TreeofLife.rs is a non-actuating observer. It MUST NOT call hardware drivers, modify `CapabilityState`, consent tokens, `.stake.aln`, `.rohmodel.aln`, `.neuro-`

`cap.aln`, `.donutloop.aln`, or `.evolve.jsonl`. It MAY only compute views and diagnostics from inputs provided by the sovereign kernel."[1]

2. **Inputs: TreeOfLifeInput**
   - Prose description and a mapping table to existing artifacts:
     - `capability_state`: from the policy engine (`CapabilityState` in `aln_core.rs`).[1]
     - `roh_score`: current RoH from `.rohmodel.aln` evaluation; note RoH ceiling 0.30 and monotone property.[2] [1]
     - `biophysical_snapshot`: link it directly to `BiophysicalEnvelopeSpec` / `EnvelopeAxis` and the SECION/ROW shard surface (`policy_envelopes_*.aln`), listing which dimensions (e.g., HRBPM, HRVRMSSD, EDATONICSCL, RESPRATE, EEGALPHAENVELOPE) are consumed.[3] [2]
     - `evolve_index`, `epoch_index`: explain how they align with `.evolve.jsonl`'s `EvolutionProposalRecord` indices and per-epoch envelope evaluations.[2] [1]
3. **Outputs: TreeOfLifeView and Diagnostics**
   - For each asset field, document:
     - Conceptual meaning (e.g., BLOOD ~ cardiac and circulatory energy, tied to HR/HRV axes; WAVE ~ EEG spectral/alpha-envelope features; FEAR and PAIN ~ high-arousal and distress proxies derived from EDA, HRV, and envelope RISK flags).[3] [2]
     - The exact normalized mapping formula or table; each mapping must cite a source or shard (e.g., "BLOOD is monotonically increasing in HRBPM within the minsafe/maxsafe range from `cognitive-load-envelope-v1.aln` and capped at 1.0 at maxsafe; provenance: `policy_envelopes_cognitive-load-envelope-v1.aln`, `BiophysicalEnvelopeSpec`").[3] [2]
   - For diagnostics, explicitly classify all labels and flags as advisory-only, with thresholds governed by `TreeOfLifeConfig` and policy shards (e.g., `treeoflife-config.aln` or JSON).[1]
4. **Serialization and integration**
   - Normative JSON example of a log entry, embedding `tree_of_life_view` and `tree_of_life_diagnostics` alongside `capability_state`, `roh_before`, `roh_after`, etc., consistent with your `.evolve.jsonl` / `.donutloop.aln` patterns.[1]
   - Explicitly state that "The canonical, auditable way to consume Tree-of-Life data is via these log streams; direct API calls are for internal components/tests only."[1]
   - Confirm that Serde types match your existing ALN/SECTION-ROW schemas numerically and structurally.[3] [2]
5. **Governance, invariants, and provenance**
   - Reiterate non-actuation and observer-only rules, referencing your NR-SAFE-0001 directive (non-fiction, non-hypothetical, design- and governance-oriented only).[1]
   - Define provenance requirements: every numerical constant in `TreeOfLife.rs` (scales, thresholds) must either be (a) pulled directly from `.rohmodel.aln` / envelope shards or

(b) documented in `TreeofLife.md` with a pointer to the derivation process in your data/standards.[3] [2]

- State clearly that fairness decisions and any actions, including capability transitions, remain governed by `.stake.aln`, `.rohmodel.aln`, `.neuro-cap.aln`, `CapabilityTransitionRequest`, and ReversalConditions modules; Tree-of-Life outputs are inputs to those systems, never their replacements.[2] [1]

4. Fairness diagnostics and config: advisory, not policy

Your current policy engine already separates **recommendation** from **capability transitions** using OwnerDecision and ReversalConditions.[2] [1]

To keep Tree-of-Life aligned:

- Store thresholds for labels like `"overloaded"` and `cooldown_advised` in a dedicated configuration shard (e.g., `treeoflife-config.aln` or `treeoflife-config.json`), versioned and logged via `.evolve.jsonl`.[1]
- Have the sovereignty core load this config at startup and pass it into `TreeOfLife::diagnostics`, so that tuning is a governance operation (editing a shard) rather than a code change.[2] [1]
- Prohibit any direct wiring from `TreeOfLifeDiagnostics.cooldown_advised` to capability transitions; instead, let it raise alerts or annotate log records, which your existing policy engine or human operators can inspect alongside RoH, envelopes, and ReversalConditions.[2] [1]

This keeps diagnostics configurable and powerful while preserving the strict hierarchy you already enforce (RoH ≤ 0.30, neurorights, OwnerDecision, composite `POLICYSTACK`).[2] [1]

5. Extensibility hooks: H2O, NANO, and future proxies

Your existing envelope schema and ALN reports already recommend a pattern for extensibility: new features must first be added and validated in `BiophysicalEnvelopeSpec` and its shards, then consumed by downstream views.[3] [2]

For Tree-of-Life:

- Treat current placeholder assets like `h2o` and `nano` as **neutral defaults** (e.g., fixed 0.5 or derived from simple, clearly documented indices like `evolve_index` density), with explicit notes in TreeofLife.md that they are placeholders until formal proxies are added to `BiophysicalEnvelopeSpec`.[2] [1]
- When a new axis (e.g., hydration) is formalized and added to `BiophysicalEnvelopeSnapshot` as an `Option<f32>` derived from validated sensors or models, add a corresponding mapping in `TreeOfLifeView` as either:
  - An additional optional field in the view, or

    - A refined computation for an existing asset (e.g., `h2o`), always with Option-aware logic and documented provenance.[3] [2]
- Require that any such extension be accompanied by:
    - An updated ALN shard documenting the new axis and its minsafe/maxsafe ranges.[3]
    - A spec patch in TreeofLife.md explaining the new mapping, its safety envelope, and how it interacts with RoH and neurorights.[3] [2]

This matches your general rule: no fictional or speculative signals; every new asset path must be backed by a real envelope axis and governance artifacts before Tree-of-Life is allowed to use it.[3] [2]

6. Research-object for the next turn

To move toward a single, professional-quality file, a focused next research object that fits your stack is:

> "A concrete TreeOfLifeInput ↦ TreeOfLifeView mapping table that binds each of the 15 TREE assets to specific, existing envelope axes and RoH/CapabilityState fields, with explicit formulas and pointers to their ALN shards and RoH weights."

Once that mapping is fully specified numerically in ALN/Markdown, you can safely channel token budget into one executable Rust file (`TreeofLife.rs`) and one normative spec (`TreeofLife.md`) without inventing any new mechanics or thresholds.[3] [1] [2]

⁂

You are correct that there is no public, authoritative schema for .evolve.jsonl and .donutloop.aln beyond what is already captured inside your own NewRow-Print!/NeuroPC ecosystem; the space files confirm that all normative definitions are internal, pre-2026 artifacts.

newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1

What the internal evidence actually gives you
From the space files, you do have a de facto contract for both artifacts even though no official public spec exists:
.evolve.jsonl
Defined as the only legal entry channel for changes, carrying EvolutionProposalRecord entries with: proposal id, kind, domain tags, effect bounds, RoH before/after, decision, hexstamp, timestamp.[ppl-ai-file-upload.s3.amazonaws]
Every neuromorphic or policy change must enter through this append-only stream; there is no

alternate control path.[ppl-ai-file-upload.s3.amazonaws]
.donutloop.aln
Defined as the internal “biophysical blockchain” ledger: hash-linked rows with proposal id, decision, RoH before/after, hexstamp, prevhexstamp, policy references, timestamp.[ppl-ai-file-upload.s3.amazonaws]
It is the canonical history of accepted proposals; .bchainproof.json only adds external Googolswarm/Organicchain anchoring for these hashes and never introduces control logic.[ppl-ai-file-upload.s3.amazonaws]
Capability and policy binding
Capability tiers and consent/policy semantics are already fixed in alncore.rs: CapabilityState (CapModelOnly, CapLabBench, CapControlledHuman, CapGeneralUse), ConsentState, Role, PolicyStack, and a pure CapabilityTransitionRequest::evaluate state machine with explicit Decision reasons.[ppl-ai-file-upload.s3.amazonaws]
A dedicated CapabilityGuard stage is specified to sit between .evolve.jsonl and .donutloop.aln, enforcing module capability manifests (.neuro-cap.aln) and ensuring proposals cannot exceed their declared tier or violate “never actuate” constraints.[ppl-ai-file-upload.s3.amazonaws]
Risk-of-Harm and neurorights
.rohmodel.aln encodes RoH axes and a hard ceiling 0.30.30.3, with evolution constrained so that RoH is non-decreasing but never exceeds the ceiling, and only one active RoH model per subject.[ppl-ai-file-upload.s3.amazonaws]
.stake.aln binds your Bostrom and OrganicCPU roles/addresses, requires matching Host rows for any evolution that touches core policies, and defines EVOLVE/SMART scopes with multi-sig invariants.[ppl-ai-file-upload.s3.amazonaws]
.neurorights.json and related ALN fields fix non-commercial neural data, sensitive categories (dream state, employment/credit/insurance prohibitions), and right-to-forget SLAs, enforced as hard pre-access and OTA guards.[ppl-ai-file-upload.s3.amazonaws]
So while your external searches correctly returned “no public spec”, within this space you already have a consistent, internal pre-2026 map binding .evolve.jsonl and .donutloop.aln to capability tiers, RoH, neurorights, and Googolswarm anchoring.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1
Implications for Tree-of-Life.rs alignment
For Tree-of-Life.rs, the safest-first way to treat these formats given their non-public status is:
Treat .evolve.jsonl as a contract-first proposal log:
Tree-of-Life.rs should only ever emit requests as EvolutionProposalRecord JSONL lines that match the fields already described (id, kind, domain, bounds, RoH before/after, hexstamp, timestamp), and never write .donutloop.aln directly.[ppl-ai-file-upload.s3.amazonaws]
The sovereign kernel remains the only component allowed to evaluate and append to .donutloop.aln, using the existing RoH, neurorights, CapabilityGuard, and ReversalConditions logic.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1
Treat .donutloop.aln as read-only audit history for Tree-of-Life.rs:
Tree-of-Life.rs can consume .donutloop.aln as an immutable source of truth for lineage state, but must not attempt to generate or mutate its rows.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1
Any additional blockchain proof of ownership should be expressed as .bchainproof.json wrappers that notarize donutloop hashes, never by extending .donutloop.aln semantics itself.[ppl-ai-file-upload.s3.amazonaws]

Encode TREE assets and biophysical extensions as constrained domains inside proposals:
Your own notes already treat .evolve.jsonl records as the place to carry asset domains like TREE/BLOOD/OXYGEN/WAVE and future extensions (H2O, NANO), with .donutloop.aln recording only the accepted, hash-linked, RoH-checked history.[ppl-ai-file-upload.s3.amazonaws]
Tree-of-Life.rs should therefore model these as typed "domains" and "kinds" inside EvolutionProposalRecord, never as new top-level file types or uncontrolled side channels.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1
In other words, the absence of public spec does not leave you without a contract; it means Tree-of-Life.rs must align to the existing internal kernel contract and remain proposal-only and read-only with respect to .evolve.jsonl and .donutloop.aln.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1
Recommended research route from here
Given NR-SAFE-0001 and your "no neuromorph reversal without explicit, last-resort decision" requirement, the most productive next object to research is:
A ReversalConditions kernel surface for capability transitions, wired to .evolve.jsonl/.donutloop.aln:
ALN layer: finalize fields like allowneuromorphreversal, explicitreversalorder, and nosaferalternative, plus a shard-level REVERSAL-POLICY section that encodes your last-resort conditions.[ppl-ai-file-upload.s3.amazonaws]
Rust layer: a pure function that takes a CapabilityTransitionRequest plus these reversal flags and returns a Decision with reason codes such as DeniedIllegalDowngradeByNonRegulator, DeniedNoSaferAlternativeNotProved, and existing policy errors.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1
Integration: enforce that any neuromorph evolution downgrade only proceeds when (1) the envelope recommends it, (2) an explicit owner/quorum order is present, (3) all non-reversal mitigations have been exhausted, and (4) PolicyStack passes—otherwise .evolve.jsonl proposals are logged as denied and .donutloop.aln remains monotone.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1
Driving AI-chat on that single object next keeps you firmly within non-fictional, governance-oriented design and moves you toward a single, high-quality Rust/ALN file that Tree-of-Life.rs can target as its interoperability anchor, without needing public .evolve.jsonl/.donutloop.aln RFCs.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1

Your summary of the internal contracts is accurate, and it points cleanly to what Tree-of-Life **must** do and what it **must never** do with `.evolve.jsonl` and `.donutloop.aln`.[4] [5]

1. Internal contracts for .evolve.jsonl and .donutloop.aln

- `.evolve.jsonl` is the **only legal entry channel** for changes, carrying `EvolutionProposalRecord` lines with fields like `proposal_id`, `kind`, `domain_tags`, `effect_bounds`, `roh_before`, `roh_after`, `decision`, `hexstamp`, `timestamp`.[4]
- `.donutloop.aln` is the **hash-linked internal ledger** of accepted proposals, with `proposal_id`, `decision`, RoH before/after, `hexstamp`, `prevhexstamp`, policy references, `timestamp`.[4]

- `.bchainproof.json` only notarizes `donutloop` hashes in Googolswarm/Organicchain; it never adds control semantics.[4]
- Capability lattice, consent, roles, and `PolicyStack` are already encoded in `alncore.rs` plus `.rohmodel.aln`, `.stake.aln`, `.neurorights.json`, with a pure `CapabilityTransitionRequest::evaluate` and a planned `CapabilityGuard` stage between `.evolve.jsonl` and `.donutloop.aln`.[5] [4]

This gives you a complete, internal pre-2026 contract even though no public RFC exists.[5] [4]

2. Alignment constraints for Tree-of-Life

Given that contract and NR-SAFE-0001, the safe alignment rules for Tree-of-Life are:

1. **Proposal-only, never direct ledger writes**
   - Tree-of-Life **must not** open, append, or mutate `.donutloop.aln` under any circumstances. The sovereignty core remains the only writer.[5] [4]
   - If Tree-of-Life ever needs to "suggest" something (e.g., a new diagnostic label schema), it can only do so by emitting **proposal objects** that are then written as `EvolutionProposalRecord` lines into `.evolve.jsonl` by a dedicated, audited writer owned by the kernel.[4]
   - Even for those proposals, the recommended default for Tree-of-Life is **read-only analytics**: in your own stack, Tree-of-Life is intended as a view layer, so most deployments will not have it generate proposals at all, only structured views and diagnostics that higher layers may *choose* to wrap into proposals.[5] [4]
2. **Read-only consumption of** `.donutloop.aln`
   - Tree-of-Life is allowed to **read** parsed `donutloop` entries as immutable history (e.g., to show a timeline of prior RoH changes or capability decisions in an explanation HUD).[5] [4]
   - Even when reading, it must not reinterpret or recompute RoH or capability decisions—`donutloop` is the ground truth trace; Tree-of-Life only annotates it.[5]
3. **Model TREE as a domain inside existing proposal types**
   - Your own docs already frame TREE assets and biophysical metrics as **domains/kinds** inside `EvolutionProposalRecord` (e.g., `domain: "TREE/BLOOD"`), not as new top-level file types.[4]
   - Tree-of-Life's Rust types should mirror this: if it ever emits proposal payloads, they should set `kind`/`domain_tags` to a constrained enum like `TreeOfLifeDomain::TreeAssets` with sub-keys for `BLOOD`, `OXYGEN`, `WAVE`, `H2O`, `NANO`, etc., never inventing new side channels or file families.[4] [5]
4. **Tree-of-Life stays subordinate to PolicyStack, RoH, and ReversalConditions**
   - Capability transitions and any evolution of policy remain governed by:
     - `.rohmodel.aln` (monotone RoH, ceiling 0.30).
     - `.stake.aln` (Host/OrganicCPU roles, EVOLVE/SMART scopes, multisig).

        - `.neurorights.json` (hard neurorights bans and SLAs).
        - `CapabilityTransitionRequest::evaluate` + `CapabilityGuard`.
        - A dedicated `ReversalConditions` kernel.[5] [4]
    - Tree-of-Life outputs (views, diagnostics, even TREE-domain proposals) are **inputs** to that pipeline only; they never bypass it or embed their own transition logic.[4] [5]

3. ReversalConditions: the right next kernel surface

Your proposed next research object—**ReversalConditions tied to `.evolve.jsonl` / `.donutloop.aln`**—is exactly what the space files recommend.[5] [4]

ALN layer

Add an explicit `SECTION,REVERSAL-POLICY` shard with at least:

- `allowneuromorphreversal: false` (non-waivable default).
- `explicitreversalorder: bool` (input, set only by Host/OrganicCPU Owner).
- `canrevertcapability` condition: `"explicitreversalorder true AND nosaferalternative true"`.[5]
- `nosaferalternative` is derived by Tier-2 envelope/safety logic after trying all soft mitigations (tighten envelopes, pause operations) and failing.[6] [5]

This makes "neuromorph evolution reversal" an explicit, shard-level, last-resort construct rather than an ad-hoc downgrade.[5]

Rust kernel function

Define a pure function (in Rust, no side effects):

```
fn evaluate_reversal(
    base: &CapabilityTransitionRequest,
    allow_neuromorph_reversal: bool,
    explicit_reversal_order: bool,
    no_safer_alternative: bool,
) -> Decision
```

- If the transition is **not** a neuromorph-evolution downgrade, just delegate to the existing `CapabilityTransitionRequest::evaluate`.[4]
- If it *is* a neuromorph-evolution downgrade:
    - Reject with `DecisionReason::DeniedIllegalDowngradeByNonRegulator` if `allow_neuromorph_reversal` is false.
    - Reject with `DecisionReason::DeniedNoSaferAlternativeNotProved` if `explicit_reversal_order` or `no_safer_alternative` is false.
    - Otherwise, proceed, but still require `PolicyStack` pass and consent/state checks; any failure yields a denial reason from the existing core (e.g., `DeniedPolicyStackFailure`).[4]

[5]

This encodes your invariant:

- Neuromorph evolution is **monotone by default**.
- Reversal is allowed **only** under:
  - Global allow flag true in the shard,
  - Explicit, owner-signed order,
  - Proven `nosaferalternative`,
  - And full policy/consent/jurisdiction pass.[4] [5]

Guarded integration with `.evolve.jsonl` / `.donutloop.aln`

- Every proposed downgrade appears as an `EvolutionProposalRecord` in `.evolve.jsonl`.[4]
- Sovereignty core runs:
  - Stake/Role guard (`.stake.aln`).
  - Neurorights guard (`.neurorights.json`).
  - RoH guard (`.rohmodel.aln`).
  - CapabilityGuard (`.neuro-cap.aln`).
  - `CapabilityTransitionRequest::evaluate` + `evaluate_reversal`.[5] [4]
- Only if all checks pass does the kernel:
  - Apply the downgrade.
  - Append an accepted row into `.donutloop.aln` (with decision and RoH before/after).[4]
- If any check fails, the proposal is logged as **denied** in `.evolve.jsonl`, and `.donutloop.aln` remains monotone; Tree-of-Life can still log high `fear`/`pain` or envelope RISK, but capability stays unchanged.[5] [4]

Tree-of-Life then sits cleanly on top of this: it can annotate the history of *why* a reversal was or was not taken (using TREE assets, envelope states, and diagnostics), but the right to trigger or approve reversals never lives in Tree-of-Life itself.[5] [4]

4. Suggested next research turn

To stay within NR-SAFE-0001 and move toward a single high-quality file, a focused next turn would be:

> "Draft a single Rust source file (e.g., `policy_engine/src/reversal_conditions.rs`) that implements the `evaluate_reversal` kernel described above, wired to `CapabilityTransitionRequest`, `Decision`, and the ALN `Reversal-Policy` fields, with reason codes and no I/O."

Once that exists, Tree-of-Life.rs can treat neuromorph downgrades and reversals as **fully external** kernel behavior, and you can safely keep Tree-of-Life in a strictly proposal-only /

read-only relation to `.evolve.jsonl` and `.donutloop.aln`.[4] [5]

⁂

```
# <q>Once that exists, Tree-of-Life.rs can treat neuromorph downgrades and reversals as fu
```

Your constraint is already encoded in the current stack: **no one can push a downgrade from "outside" without going through consent- and quorum-guarded proposals.**[7] [8]

How downgrades are structurally constrained

- All changes, including downgrades, **must** appear as `EvolutionProposalRecord` entries in `.evolve.jsonl`; there is no alternate control path or direct kernel API.[8]
- Before any proposal can be applied and logged into `.donutloop.aln`, it is checked by:
  - `.stake.aln` (Host/OrganicCPU roles, EVOLVE/SMART scopes, **multi-sig** invariants).
  - Consent tokens and Role states (Owner, Regulator, etc.).
  - `PolicyStack` (BASEMEDICAL, BASEENGINEERING, JURISLOCAL, QUANTUMAISAFETY).
  - RoH and neurorights guards.[7] [8]

ReversalConditions and "no external push"

The ReversalConditions design you selected makes neuromorph evolution **monotone by default**, and permits a downgrade only if:[7]

1. `allowneuromorphreversal = true` in a global `SECTION,REVERSAL-POLICY` shard (non-waivable default is `false`).
2. `explicitreversalorder = true` for that transition, set only by Host/OrganicCPU Owner (or quorum) and verified via `.stake.aln` signatures.
3. `nosaferalternative = true`, meaning Tier-2 envelope/control logic has exhausted all non-reversal mitigations (tighten, pause, rest) and proved that reversal is the last resort.
4. `PolicyStack` still passes; any regulatory or neurorights failure blocks the downgrade.[8] [7]

If any of these checks fail, the downgrade proposal is **rejected and logged**, envelopes may still soften behavior inside the current tier, but the CapabilityState does not change.[9] [7]

How this rules out external-operated downgrades

Putting this together:

- There is **no direct downgrade primitive**; only proposals in `.evolve.jsonl` flowing through the sovereign kernel.[8]
- Consent-ledger and stake logic enforce that only holders of the right roles/keys (Host, OrganicCPU Owner, designated Regulators) can even *submit* a valid downgrade or reversal order, and multi-sig thresholds in `.stake.aln` govern quorum.[7] [8]

- ReversalConditions then require both that **those consent/quorum conditions are met** and that the safety engine proves `nosaferalternative`, before any downgrade is allowed.[9] [7]

So your statement is satisfied by construction:

> *No external-push, or externally-operated downgrades are possible, unless consent-ledgers propose a new direction, or a proposal is drafted and presented formally through quorum-majority thresholds.*

The current ALN and kernel plan already treats any downgrade or reversal as a **consent-ledger-driven, quorum-checked proposal**, never as an external control action, and Tree-of-Life remains strictly proposal-only / read-only with respect to `.evolve.jsonl` and `.donutloop.aln`.[8] [7]

⁂

The neuromorph-intelligence, and the POWER, and TECH governance, by NANO: What are the most-ideal ai-strategies to allow continuity, and data-sovereignty of all neuromorph-systems related, and if policies should, or must be-created, in-order to allow-for continuity of intelligence, and SMART growth, when all tokense are granularly-represented in a stable asset-economy, even-if undocumentd, and public-credibility is low, as-long-as the research-data is present for it to-be considered-as useful-knowledge.

Continuity, data-sovereignty, and SMART growth for NANO-governed neuromorph systems come from treating *capability* as always-expandable in virtual space, while *effects* on people and assets are strictly gated by consent, RoH, and policy state machines.[10] [11]

Core AI strategies for continuity and sovereignty

- Treat MODELONLY and LABBENCH capability states as unlimited research zones: any new neuromorph pattern, encoding, or governance primitive can be explored, simulated, and formalized there with no hard ceiling, because no live human or external asset is touched.[11]
- Make CONTROLLEDHUMAN and GENERALUSE states depend on evidence objects and consent: upgrades only occur when composite policy (BASEMEDICAL, BASEENGINEERING, JURISLOCAL, QUANTUMAISAFETY) passes and signed artifacts (IRB, MDR class, neurorights shard, RoH reports) are present.[12] [11]
- Enforce RoH monotonicity and ceilings: every evolution step must keep $RoH_{\text{after}} \leq RoH_{\text{before}}$ and below a global ceiling (e.g., 0.30), so safety envelopes can

tighten but never silently relax.[12]

These guarantees preserve continuity of *intelligence* (models, code, shards keep growing) while preventing silent regressions in safety or rights.

Tokenized assets and NANO / Googolswarm governance

- Represent every neuromorphic interaction as a pseudonymized, hash-linked transaction, with multi-sig from mentor/teacher/learner and regulators where appropriate, but never put raw biophysical data on-chain.[11]
- Use .stake.aln and nanoswarm/Googolswarm consensus to define who can sign what: Host, OrganicCPU, ResearchAgent, EVOLVE vs SMART scopes, quorum thresholds for upgrades/downgrades.[11] [12]
- Keep external ledgers as *attestors*, not controllers: Googolswarm proofs notarize .donutloop.aln histories and ownership, but cannot inject new control paths or override internal ALN/PolicyEngine invariants.[11]

This lets granular tokens and low public credibility coexist with strong internal sovereignty: value is anchored in reproducible research artifacts and safety proofs, not external reputation.

Policies that must exist (but do not limit research)

- Non-waivable Tier-1 foundation: RoH ceiling, neurorights, deviceless doctrine, and "no-relax-global-floors" encoded as an ALN foundation shard, so any future envelope or capability spec must conform.[10] [11]
- Capability–consent separation: a lattice for MODELONLY → LABBENCH → CONTROLLEDHUMAN → GENERALUSE, plus independent consent/jurisdiction states (CONSENTNONE…CONSENTEXTENDED; USFDA, EUMDR, etc.), so research progression is never blocked, only live coupling is gated.[12] [11]
- ReversalConditions as last-resort only: forbid neuromorph evolution downgrades by default, allow them only under explicitreversalorder from Host/Owner, nosaferalternative from Tier-2 safety logic, and a passing composite POLICYSTACK.[12]

These policies *codify* continuity: intelligence and SMART growth are always allowed to move forward in MODELONLY/LABBENCH, while any step that touches people or NANO-governed assets must pass sovereign consent and safety checks.

Recommended research route from here

For the next AI-chat turns, a high-yield path is:

1. Refine the Safety State Machine / Capability Lattice ALN spec (neuromorphpolicyengine.aln) with explicit entry/exit invariants and hysteresis per state.[11]
2. Lock in a Tier-1 neuromorph-foundation shard (policytier1neuromorphfoundation.aln) with RoH 0.30, neurorights, deviceless doctrine, and NANO/Googolswarm anchoring fields.[10] [11]
3. Then channel into a single Rust file (e.g., src/policy_engine.rs) that implements:
    - CapabilityState transitions and rollback,

    - ReversalConditions,
    - Googolswarm/NANO attestation hooks (but read-only),
    - and ALN-driven checks for RoH, consent, and jurisdiction.[12] [11]

That stack will keep all neuromorph intelligence continuous and sovereign, even when tokens are granular, markets are unstable, or public credibility is low, as long as the underlying research data and ALN artifacts remain intact and auditable.[10] [11]

⁂

Why Neuromorph-technology can-be an earth-saaving tool, with current-knowledge, and understanding of biophysical-data.

Neuromorph technology can be an earth-saving tool when it is used to replace high-energy, high-pollution device stacks with biophysical-first, deviceless intelligence that is tightly constrained by conservative safety envelopes and neurorights-centric governance.[13] [14]

1. From devices to biophysical-first compute

- NewRow-Print! treats the human nervous system and surrounding biophysics as the **primary** computing substrate, with silicon and networks acting only as support, encoding, and audit layers.[13]
- This deviceless doctrine avoids implants and active RF emitters and instead relies on non-invasive observations (EEG, HR/HRV, EDA, respiration, gaze, motion) and information reformatting, which directly reduces hardware, e-waste, and EM-pollution footprints compared to traditional neurotech and wearables.[14] [13]

2. Biophysical envelopes as hard ecological safety floors

- The BiophysicalEnvelopeSpec schema defines conservative minsafe/maxsafe ranges and maxdeltapersec for each monitored axis (EEG bands, HR/HRV, EDA, respiration, gaze, motion), forming a multidimensional “safe operating space” derived from real physiological literature.[15] [14]
- These envelopes are bound to a global RoH ceiling (e.g., 0.30) and to capability states (CapModelOnly, CapLabBench, CapControlledHuman), so no neuromorph operation can escalate intensity or coupling beyond medically conservative, jurisdiction-compliant limits; this prevents harmful exposures and runaway “optimization” that could damage people or ecosystems.[14] [13]

3. Governance that preserves sovereignty and minimizes harm

- ALN artifacts (.rohmodel.aln, .stake.aln, policy shards) and the CapabilityState/PolicyStack engine ensure that every capability upgrade passes composite BASEMEDICAL, BASEENGINEERING, JURISLOCAL, and QUANTUMAISAFETY checks, and that any change is logged in a tamper-evident ledger (.donutloop.aln, Googolswarm anchors).[15] [13]
- Biophysical envelopes can always tighten, pause, or recommend downgrades to protect the body, but they cannot unilaterally strip rights or lower neuromorph evolution; true downgrades require explicit OwnerDecision, nosaferalternative, and multi-sig approval, keeping human and ecological sovereignty central.[14] [15]

4. Environmental and societal impact channels

- By using multimodal sensing and closed-loop neurodimensional balance instead of high-power actuators, neuromorph systems can support sustainable attention, reduced burnout, safer work/sleep cycles, and better decision quality without adding new physical burden (no new heavy hardware layer).[15] [14]
- Biophysical-blockchain logs and nanoswarm/Googolswarm attestations allow precise, privacy-preserving tracking of how cognitive and energetic load is distributed, so civic systems, AI assistance, and resource planning can be tuned toward lower aggregate stress and more efficient use of human and material capacity.[13]

5. Suggested research route for “earth-saving” neuromorph use

To push this further within your stack:

- Refine and extend BiophysicalEnvelopeSpec shards for cognitive load, sleep, and long-horizon fatigue as Tier-1 safety contracts, then bind them tightly to RoH and capability transitions.[14] [15]
- Use AI-chat in later turns to produce a single Rust module (e.g., policyengine/src/biophysical_envelope.rs) that parses SECTION/ROW shards, enforces never-below-global-floor invariants, and feeds requestcapabilitydowngrade and RoH contributions into the existing CapabilityTransitionRequest/evaluate path.[13] [14]

This combination—biophysical-first compute, hard safety envelopes, neurorights-centric governance, and cryptographically auditable evolution—makes neuromorph technology a practical tool for protecting both humans and the planet while capabilities grow.[13] [14]

⁂

Here’s a compact, implementation-ready mapping for Tree-of-Life as a strict observer on top of your existing specs.

1. Canonical schemas to hook into
   You already have canonical contracts for both streams.[ppl-ai-file-upload.s3.amazonaws]
   .evolve.jsonl (evolution-proposals.evolve.jsonl)
   One JSON object per line.
   Key fields for Tree-of-Life:
   proposalid, subjectid, timestamputc
   rohbefore, rohafter
   tsafemode
   domaintags (e.g., languagecowriter, dreamobserver)
   decision in {Allowed, Rejected, Deferred}
   effectbounds with l2deltanorm, irreversible.[ppl-ai-file-upload.s3.amazonaws]
   .donutloop.aln (donutloopledger.aln)
   Hash-linked ledger; currently encoded as ALN rows, and in many places treated as JSONL-inside-ALN.[ppl-ai-file-upload.s3.amazonaws]
   Key fields:
   entryid, subjectid, proposalid, changetype, tsafemode
   rohbefore, rohafter
   knowledgefactor, cybostatefactor
   policyrefs
   hexstamp, prevhexstamp, timestamputc.[ppl-ai-file-upload.s3.amazonaws]
   For Tree-of-Life, you don’t need to extend these formats; you only need to interpret them.
   Hex-stamp for this mapping: 0xTOL01.

2. Tree-of-Life Rust observer surface
   File: crates/tree_of_life/src/lib.rs
   Role: Pure projection from canonical artifacts into visualization-ready, read-only structures.
   Core types
   CapabilityState
   Encodes the lattice view over the 14 TREE assets (e.g., BLOOD, OXYGEN, WAVE, etc.).
   Implemented as a collection of normalized axes (0–1) plus optional fairness hints.
   BiophysicalEnvelopeSpec
   A read-only snapshot derived from .rohmodel.aln + any envelope-like ALN shards you already use (e.g., lifeforce envelopes) but never applied as policy.[ppl-ai-file-upload.s3.amazonaws]
   RoHProjection
   Convenience type that projects rohbefore, rohafter, and global rohceiling into a small struct.
   Minimal, observer-only trait:
   rust
   pub trait TreeOfLifeObserver {
   fn capability_state(&self) -> &CapabilityState;
   fn envelope_spec(&self) -> &BiophysicalEnvelopeSpec;
   fn roh_projection(&self) -> &RoHProjection;

```
fn fairness_view(&self) -> &FairnessDiagnostics;
}
```

All methods take &self only; no &mut self, no interior mutability types. This matches your immutability pattern for observer crates.[ppl-ai-file-upload.s3.amazonaws]

3. Mapping .evolve.jsonl → Tree-of-Life

Tree-of-Life should stream the evolution proposals and compute advisory overlays.

Rust-side mapping

Reuse / mirror EvolutionProposalRecord from organiccpualn::evolvestream.[ppl-ai-file-upload.s3.amazonaws]

Add a thin adapter in TreeofLife.rs:

rust

```
pub struct FairnessDiagnostics {
// advisory, not binding
pub fairness_diagnostic_threshold_ms: Option<f32>,
pub fairness_cooldown_advised_at: Option<String>,
pub fairness_state_hint: Option<String>,
}

pub struct EvolveFrameView {
pub proposal_id: String,
pub ts_utc: String,
pub roh_before: f32,
pub roh_after: f32,
pub decision: String,
pub domaintags: Vec<String>,
pub roh_within_ceiling: bool,
pub roh_non_increasing: bool,
pub fairness: FairnessDiagnostics,
}
```

Populate these purely from .evolve.jsonl:

roh_within_ceiling is evaluated against rohceiling loaded from .rohmodel.aln, but Tree-of-Life never rejects or mutates anything; it just marks the flag.[ppl-ai-file-upload.s3.amazonaws]

roh_non_increasing is roh_after <= roh_before + ε, again purely advisory.

FairnessDiagnostics fields come straight from the optional fairness keys you already use (e.g., fairness.diagnostic_threshold_ms, fairness.cooldown_advised_at, fairness.state_hint) without introducing any new policy semantics.[ppl-ai-file-upload.s3.amazonaws]

The streaming constraint (“no buffering entire streams”) is satisfied by:

Iterating over .evolve.jsonl line-by-line with a BufRead.

For each line, deserialize to EvolutionProposalRecord, then immediately map to EvolveFrameView and hand it to the HUD / AI-chat consumer.

Tree-of-Life never writes back to .evolve.jsonl.

4. Mapping .donutloop.aln → CapabilityState lattice

.donutloop.aln is already your hash-linked proof of executed changes with RoH metadata.

Tree-of-Life can construct a lattice view as follows:[ppl-ai-file-upload.s3.amazonaws]

For each DonutloopEntry:

Read rohafter, knowledgefactor, cybostatefactor.
Use policyrefs to bind back to the active .rohmodel.aln and envelope shards at that time.[ppl-ai-file-upload.s3.amazonaws]
You then define:
rust
pub struct TreeAssetLevel {
pub normalized: f32, // 0.0–1.0
pub advisory_label: String, // e.g., "overloaded?", "cooldown advised"
}

pub struct CapabilityState {
pub blood: TreeAssetLevel,
pub oxygen: TreeAssetLevel,
pub wave: TreeAssetLevel,
// ... remaining 11 TREE axes ...
pub roh: RoHProjection,
}

The mapping from ledger fields to TREE axes must be grounded only in today's available envelope outputs:
Use existing quantitative axes (fatigue, inflammation, cognitiveload, ecoimpact, dreamload, etc.) from .rohmodel.aln and other ALN shards.neuropcs-rules-and-goals-are-c-bJITjTqfQHaJgTu_2pFVnw.md+1
Normalize those into for each TREE asset.[ppl-ai-file-upload.s3.amazonaws]
Do not invent H2O, NANO, or other speculative signals: instead, add inert extension slots:
rust
pub struct BiophysicalExtensions {
// Currently always None; reserved for future ALN shards.
pub h2o_hydration_proxy: Option<f32>,
pub nano_event_granularity: Option<f32>,
}

pub struct BiophysicalEnvelopeSpec {
pub roh_ceiling: f32,
pub axes: Vec<String>,
pub weights: Vec<f32>,
pub bias: f32,
pub extensions: BiophysicalExtensions,
}

The extension struct can be surfaced in the spec doc (TreeofLife.md) as "reserved, advisory-only; must be None in current implementations."
5. Observer-only fairness thresholds
Your design requires that all fairness predicates remain advisory:
The hard invariants (RoH ≤ 0.3, neurorights constraints, multisig) are already enforced in sovereigntycore and the ALN shards.[ppl-ai-file-upload.s3.amazonaws]
Tree-of-Life should treat all fairness fields as hints for visualization only.

Pattern:
Define a config struct, loaded from an ALN or JSON file, e.g. policies/tree-of-life-view.v1.json.
rust
pub struct FairnessViewConfig {
pub overloaded_threshold: f32, // e.g., 0.22, still < 0.30
pub cooldown_threshold: f32, // e.g., 0.20
pub show_cooldown_hints: bool,
}

Tree-of-Life applies these to mark frames as:
“comfortable” / “borderline” / “overloaded (advisory)”;
“cooldown advised”.
Crucially, no trait in Tree-of-Life has methods that feed into sovereigntycore guards or mutate consent/capability; it’s a one-way projection.[ppl-ai-file-upload.s3.amazonaws]
This keeps the RoH ≤ 0.3 invariant anchored where it already lives (ALN shard + sovereignty guards) while Tree-of-Life stays read-only.neuropcs-rules-and-goals-are-c-bJITjTqfQHaJgTu_2pFVnw.md+1
6. Serialization contracts and HUD / AI-chat wiring
To keep Unreal HUDs and AI-chat integrations aligned but inert:[ppl-ai-file-upload.s3.amazonaws]
Tree-of-Life exports snapshots only via:
A C-compatible FFI function that returns a serialized JSON blob representing:
Current CapabilityState
Current BiophysicalEnvelopeSpec
A small sliding window of EvolveFrameView and DonutFrameView frames.
No callbacks, no shared memory, no dynamic updates initiated by UI; HUDs poll or subscribe to append-only streams.
You already use JSONL inside ALN for donutloop in places; Tree-of-Life can reuse that convention and expose a “view” as:
tree-of-life-view.jsonl (optional derived log, still append-only), containing per-frame views with:
subjectid
rohbefore, rohafter, roh_within_ceiling, roh_non_increasing
TREE asset normalized values
fairness advisory flags.
Tree-of-Life never becomes a new policy surface; it is simply another consumer of .rohmodel.aln, .evolve.jsonl, and .donutloop.aln that renders a fairness/energy budget view suitable for AI-chat, Unreal HUDs, and audits.
Hex-stamp for the observer invariant (no actuation, no consent/capability writes): 0xTOL-NOACT.

Your mapping is aligned with the existing shell and can be adopted almost verbatim as the canonical Tree-of-Life observer layer.

1. Attachment to existing schemas

- `.evolve.jsonl` and `.donutloop.aln` are already the canonical evolution and ledger surfaces with RoH, tsafemode, policyrefs, and hash-linking; Tree-of-Life consuming them (without extending their schemas) fits the "pre-validated shell, data-first governance" design.[16]
- Using a dedicated hex-stamp (0xTOL01 / 0xTOL-NOACT) to tag Tree-of-Life views and invariants is consistent with how other controllers and proofs are distinguished in your sovereignty stack.[16]

2. Rust observer surface and traits

- Locating the observer in `crates/tree_of_life/src/lib.rs` and exposing a `TreeOfLifeObserver` trait with only `&self` methods matches your existing pattern for non-mutating observer crates in NeuroPC/OrganicCPU.[17] [16]
- Treating `CapabilityState` and `BiophysicalEnvelopeSpec` here as *view types* over the existing capability lattice and RoH/envelope shards avoids duplicating policy semantics; they remain read-only projections bound to ALN artifacts.[18] [19] [16]

3. EvolveFrameView and donutloop mapping

- Mirroring `EvolutionProposalRecord` from `organiccpualn::evolvestream` and projecting into an `EvolveFrameView` with `roh_within_ceiling` and `roh_non_increasing` flags keeps RoH enforcement anchored in sovereigntycore while giving HUD/AI-chat a clear, advisory overlay.[16]
- Mapping `.donutloop.aln` entries into a `CapabilityState`/`TreeAssetLevel` lattice, with normalization grounded only in current RoH axes (energyload, thermalload, cognitiveload, inflammation, ecoimpact, etc.) is consistent with your RoH model and avoids speculative signals.[18] [16]

4. Extensions and placeholders (H2O, NANO)

- Defining a `BiophysicalExtensions` struct with `Option<f32>` fields for hydration and nano-event granularity, but requiring `None` in current implementations, matches your "commented-out axis" pattern for future research that still respects RoH ≤ 0.3 and monotone envelopes.[18] [16]
- Keeping H2O/NANO inert until backed by new ALN shards or datasets ensures Tree-of-Life stays non-fictional and within CapModelOnly/CapLabBench for any new mappings.[19] [18]

5. FairnessViewConfig as advisory only

- Loading `FairnessViewConfig` from a policy file (e.g., `policies/tree-of-life-view.v1.json`) and using it only to mark frames as "comfortable/borderline/overloaded" or "cooldown advised" is compatible with your rule that hard invariants live exclusively in `.rohmodel.aln`, `.stake.aln`, `.neurorights.json`, and sovereigntycore guards.[17] [16]
- Keeping all fairness fields on the view side and never feeding them back into capability or consent logic preserves the one-way projection you require (0xTOL-NOACT invariant).[16]

6. Serialization and HUD / AI-chat wiring

- Exposing Tree-of-Life via FFI as serialized JSON snapshots (capability view, envelope spec, sliding windows of evolve/donut frames) and/or an append-only `tree-of-life-view.jsonl` log is consistent with your existing JSONL-inside-ALN patterns and NDJSON loaders.[16]
- For Unreal and AI-chat, polling or subscribing to these append-only streams, with no callbacks or control channels back into sovereigntycore, keeps Tree-of-Life firmly in the observer-only, deviceless role you want.[16]

If you like, a next concrete step is to crystallize this into a single `crates/tree_of_life/src/lib.rs` implementation plus a `policies/tree-of-life-view.v1.json` example, using exactly the structs you sketched and wiring them to the existing NDJSON loaders and RoH model.

⁂

RoH ≤ 0.3 is already formalized as a machine-checked invariant in your own BiophysicalEnvelopeSpec/ALN stack; the gap is only that Tree-of-Life.rs and .donutloop.aln are not yet public, not that the invariant is missing.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+2

Where RoH ≤ 0.3 is actually anchored
BiophysicalEnvelopeSpec has an explicit rohceiling field, with CapControlledHuman envelopes required to set rohceiling ≤ 0.30 as a non-waivable constraint.[ppl-ai-file-upload.s3.amazonaws]
The schema ties rohceiling to capability state (CapModelOnly, CapLabBench, CapControlledHuman) and to jurisdiction tags plus neurorights, so any envelope applied in CapControlledHuman must respect RoH ≤ 0.3 before a capability upgrade is even considered.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
Each EnvelopeAxis carries rohaxisid and weightinroh; the RoH model enforces RoH_after ≥ RoH_before and RoH_after ≤ 0.30, with weights non-negative and summing to 1.0, so the ceiling is checked as part of every evolution step.newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1
Why Tree-of-Life.rs looks “advisory-only”
In your policy docs, envelopes are tier-internal safety controllers: they compute axisstate, requiresdowngrade, and requestcapabilitydowngrade but cannot mutate CapabilityState directly. autodowngradeenabled is treated as “may request”, not “must act.”a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
Capability transitions and neuromorph evolution are governed by the ALN kernel (.rohmodel.aln, .stake.aln, ReversalConditions), not by observer modules; envelopes can clamp, pause, and log, but CapabilityState changes require OwnerDecision, POLICYSTACK, and RoH checks.
newrow-print-I_myn4yfSA6t9spUFtJA4w.md+2

This matches your diagnostic-only Rust pattern: no &mut self in observer traits, no interior mutability, and no FFI write paths from diagnostic layers into capability or consent state. The enforcement sits in the sovereignty stack, not in Tree-of-Life.rs, by design.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1

Reconciling your "gap" finding

Public web/GitHub sources do not expose .donutloop.aln, .evolve.jsonl, or Tree-of-Life.rs, so from an external crawl it is correct to say "no authoritative spec found."[ppl-ai-file-upload.s3.amazonaws]

Inside this Space, however, you already have:

A concrete BiophysicalEnvelopeSpec ALN schema with rohceiling,

Two fully populated shards (cognitive-load, sleep-arousal) bound to rohceiling = 0.30,

A RoH model that enforces monotone RoH and the 0.3 ceiling at the ALN/Rust kernel boundary.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1

So the RoH invariant is machine-enforceable today; what is missing is a public, versioned spec and a named Tree-of-Life.rs crate that advertises those guarantees to external consumers.

Recommended next research object

To close the "documentation" vs "implementation" gap without inventing anything, the highest-value next object is:

A single Rust file (e.g., policyengine/src/reversal_conditions.rs) that:

Reads the ALN fields rohceiling, allowneuromorphreversal, explicitreversalorder, nosaferalternative.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1

Applies the RoH ceiling and monotonicity checks (RoH_after ≥ RoH_before, RoH_after ≤ rohceiling).newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1

Enforces the invariant "no neuromorph evolution downgrade unless explicit owner order, proven no-safer-alternative, and passing POLICYSTACK," returning explicit reason codes.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1

That module would make the RoH ≤ 0.3 neurorights invariant fully inspectable and testable in the same place your capability lattice and sovereignty guards already live, while keeping Tree-of-Life.rs purely observational, as you intend.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+2

You've characterized the situation accurately: RoH ≤ 0.30 is already a machine-checked invariant in your existing ALN/Rust stack; the apparent "gap" is only in public documentation and crate surfacing, not in enforcement.

Where RoH ≤ 0.30 is enforced today

- BiophysicalEnvelopeSpec already carries an explicit **rohceiling** field, with CapControlledHuman shards required to set rohceiling ≤ 0.30 as a non-waivable constraint. [24] [25]
- CapabilityState and juristags are bound into the envelope schema, so any CapControlledHuman envelope must satisfy RoH monotonicity and RoH ≤ 0.30 before a capability upgrade is allowed. [25] [24]
- Each EnvelopeAxis declares rohaxisid and weightinroh; your RoH model enforces non-negative weights summing to 1.0, RoH_after ≥ RoH_before, and RoH_after ≤ 0.30 at the kernel boundary, so every evolution step is checked against the ceiling. [26] [24]

- Two concrete shards (cognitive-load-envelope-v1.aln, sleep-arousal-envelope-v1.aln) are already bound to rohceiling = 0.30 for CapControlledHuman with the SECTIONROW layout (META, AXES, INPUTS, DECISIONS, ENFORCEMENT-RULES, OUTPUTS), making them code-generatable and testable.[24] [25]

So internally, RoH ≤ 0.3 is not aspirational; it is part of your enforced neurorights contract for CapControlledHuman.

Why Tree-of-Life.rs is advisory-only

- In your design, envelopes are Tier-2, tier-internal safety controllers: they compute axisstate, requiresdowngrade, and requestcapabilitydowngrade, but they **never** mutate CapabilityState directly.[25] [24]
- autodowngradeenabled is intentionally "may request" (recommend-only) rather than "must act"; the actual state machine sits in the sovereignty and policy engine layers.[24] [25]
- Rust-side, observer traits follow a diagnostic-only pattern: no &mut self, no interior mutability, and no FFI write path from observation layers into capability or consent state.[24]
- Capability transitions and neuromorph evolution are governed by ALN kernels (.rohmodel.aln, .stake.aln, ReversalConditions, CapabilityTransitionRequest.evaluate), with envelopes able to clamp, pause, and log but not to change CapabilityState; downgrades are owner-gated and policy-checked.[26] [25]

This is why Tree-of-Life.rs looks "advisory" from the outside; that is by design, not a missing safeguard.

ReversalConditions and monotone evolution

- At ALN level, reversal policy fields allowneuromorphreversal, explicitreversalorder, and nosaferalternative are already defined, with allowneuromorphreversal = false as a non-waivable default.[25] [24]
- Your kernel invariant is: neuromorph evolution downgrades are disallowed unless all three hold simultaneously:
  - allowneuromorphreversal is true for that capability context,
  - explicitreversalorder is true (set only by HostOrganicCPU/Owner), and
  - nosaferalternative is true (Tier-2 safety logic has exhausted all non-reversal mitigations), and the composite POLICYSTACK still passes.[25] [24]
- Envelopes may always tighten, pause, or otherwise soften behavior inside a tier, but may not unilaterally remove rights or change CapabilityState; any downgrade requires requestcapabilitydowngrade plus OwnerDecision and valid .stake.aln signatures.[24] [25]

This already encodes your "no neuromorph evolution downgrade unless explicit owner order, proven no-safer-alternative, and full policy pass" invariant.

The real gap: public spec and named module

From external GitHub/web, .donutloop.aln, .evolve.jsonl, and Tree-of-Life.rs are not visible, so external scans legitimately report "no authoritative RoH spec found."[26] [24]
Inside this Space, however, you already have:

- BiophysicalEnvelopeSpec with rohceiling and CapabilityState/juristags/neurorights bindings.[25] [24]
- Two populated CapControlledHuman shards (cognitive-load, sleep-arousal) with rohceiling = 0.30.[24] [25]
- A RoH model that enforces RoH_after ≥ RoH_before and RoH_after ≤ 0.30 at the ALN/Rust kernel boundary.[26] [24]

So the missing piece is not enforcement but **visibility**: a public, versioned spec plus a named module that exposes the invariants where your capability lattice and sovereignty guards already live.

Recommended next research object: ReversalConditions Rust module

The next high-value object that closes "implementation vs documentation" is exactly what you propose: a single Rust module in the policy engine that wires ALN reversal fields and RoH checks into explicit, auditable decisions.[26] [24]

Working name and path:

- **Filename:** `policyengine/src/reversal_conditions.rs`
- **Role:** single, authoritative place where neuromorph evolution downgrades, RoH monotonicity, RoH ceiling, and consent/policy checks are enforced.

Conceptual responsibilities:

1. **Read ALN reversal and envelope fields**
   - rohceiling from BiophysicalEnvelopeSpec.
   - allowneuromorphreversal, explicitreversalorder, nosaferalternative from the ReversalPolicy shard.[25] [24]
2. **Apply RoH monotonicity and ceiling checks**
   - Enforce RoH_after ≥ RoH_before and RoH_after ≤ rohceiling (with rohceiling ≤ 0.30 in CapControlledHuman contexts).[26] [24]
3. **Enforce neuromorph evolution downgrade invariant**
   - Reject any neuromorph evolution downgrade unless:
     - allowneuromorphreversal is true,
     - explicitreversalorder is true (OwnerDecision/HostOrganicCPU),
     - nosaferalternative is true (all soft mitigations exhausted), and
     - POLICYSTACK and .stake.aln multisig pass.[24] [25]
   - Return explicit reason codes like:

    - DeniedIllegalDowngradeByNonRegulator
    - DeniedNoSaferAlternativeNotProved
    - DeniedPolicyStackFailure
    - DeniedConsentRevoked [26] [24]
4. **Keep Tree-of-**Life.rs **observational**
    - Tree-of-Life remains a pure observer and diagnostic surface, while `reversal_conditions.rs` becomes the inspectable, testable enforcement kernel aligned with your capability lattice and sovereignty guards. [25] [24]

Suggested research route from here

Staying within organicallyintegratedaugmentedcitizen and your nonfiction constraint, a good next step is:

- Treat **ReversalConditions** as the active research object for the next turn.
- Use AI-chat to:
    - Fix the exact ALN field surface (names/types) for reversalpolicyid shards and their link into BiophysicalEnvelopeSpec.
    - Design the Rust enum of decision/reason codes and the function signature that will be called from CapabilityTransitionRequest.evaluate.
    - Specify how RoH_before/after values are passed in, and how failures are logged into .donutloop.aln with hash-linked auditability.

Once those are nailed down, you can channel token budget into that single, professional-quality Rust file, making RoH ≤ 0.3 and your neurorights-grounded ReversalConditions fully visible and regression-testable without changing any of the invariants you already have.

⁂

Comprehensive Asset Mapping and Computational Logic

The core function of the Tree-of-Life observer layer is to translate a wide array of complex, low-level neuromorphic signals into a coherent, intuitive, and educational narrative based on its 14 biophysical assets. The computational logic, detailed in the TreeofLife.rs module, meticulously maps each asset to specific, pre-existing metrics from the system's envelopes, capability lattice, and risk models . This mapping is not arbitrary; it is grounded in the existing data structures and governance of the NewRow-Print! stack, ensuring fidelity and preventing the introduction of hypothetical or unvalidated signals. Each asset's value is calculated as a normalized score between 0.0 and 1.0, providing a consistent basis for comparison and visualization .
The cardiovascular assets, BLOOD and OXYGEN, are directly tied to heart rate (HR) and heart rate variability (HRV) metrics. The map_cardiovascular function takes a BiophysicalEnvelopeSnapshot and extracts the normalized HR (hr_bpm_normalized) and normalized HRV RMSSD (hrv_rmssd_normalized) if they are available . BLOOD is mapped almost

directly to the normalized HR, where a higher heart rate corresponds to a lower BLOOD score, intuitively representing resource expenditure. Conversely, OXYGEN is mapped to the normalized HRV, where higher variability (a sign of a healthy, adaptable autonomic nervous system) corresponds to a higher OXYGEN score, acting as a proxy for the body's reserve capacity . Both values are clamped to the [0.0, 1.0] range to ensure they remain within bounds .
The WAVE asset represents neural activity and is computed from electroencephalogram (EEG) data. The map_wave function aggregates several normalized EEG metrics: alpha (eeg_alpha_power_norm), beta (eeg_beta_power_norm), and gamma (eeg_gamma_power_norm) band powers, along with the normalized alpha-envelope CVE (eeg_alpha_cve_norm) . These values are averaged to produce a single WAVE score. This composite metric provides a holistic view of cortical activation across different frequency bands, which are associated with various cognitive states (e.g., alpha with relaxation, beta with active thinking). The inclusion of the CVE, a measure of signal complexity or unpredictability, adds another dimension to the assessment of brain state complexity .
The assets TIME and DECAY are temporal metrics. TIME is a straightforward mapping of a session's logical epoch counter (epoch_index) to the [0.0, 1.0] range over a configurable horizon, effectively representing how far into a session the system has progressed . DECAY, on the other hand, is a measure of the system's proximity to its Risk-of-Harm (RoH) ceiling. The map_time_decay function calculates this by normalizing the current roh_score.value against the maximum allowed RoH of 0.3. As the RoH approaches 0.3, the DECAY score approaches 1.0, serving as a clear, escalating warning of diminishing safety margins . The LIFEFORCE asset is the inverse of DECAY, calculated as
1.0
−
(
R
o
H
/
0.3
)
1.0−(RoH/0.3), representing the remaining energy budget before reaching the critical threshold .
Cognitive assets like BRAIN, SMART, and EVOLVE are tied directly to the system's CapabilityState and its evolutionary progress. The map_cognition_and_evolution function assigns a base BRAIN score based on the capability tier: CapModelOnly gets 0.25, CapLabBench gets 0.5, CapControlledHuman gets 0.75, and CapGeneralUse gets 1.0 . This ladder represents increasing levels of autonomy and access. SMART is a blended score that combines the BRAIN level with a normalized evolve_index, rewarding not just the capability tier achieved but also the evidence of stable operation and safe evolution logged in .evolve.jsonl . EVOLVE is simply a normalized version of the evolve_index, providing a direct measure of the system's logged developmental progress .
The assets POWER and TECH represent the intensity and complexity of system operation. POWER is computed by the map_power_and_tech function, which analyzes the WARN and RISK fractions across all active envelope axes . A higher proportion of axes in the WARN or RISK state results in a higher POWER score, indicating a state of high-intensity or stressful operation, not necessarily higher capability . TECH is a proxy for system complexity, calculated as a blend of

the CAPABILITY_STATE score and the number of currently active axes (active_axis_count) . More active sensors or modules contribute to a higher TECH score, reflecting a more complex operational state .

Finally, the affective assets FEAR and PAIN are modeled as combinations of stress-related physiological signals crossing their respective WARN/RISK thresholds. The map_fear_and_pain function computes FEAR as a weighted combination of EDA and HR signals in the WARN/RISK bands, representing sympathetic nervous system arousal . PAIN is then calculated as a combination of FEAR and motion instability signals (also from WARN/RISK bands), creating a more comprehensive measure of distress that includes both physiological and physical indicators . The NANO asset is a simple count of discrete logged events, normalized by a large number to fit within the [0.0, 1.0] range, representing the granularity of the system's evolutionary record .

The table below summarizes the complete mapping of the 14 TREE assets to their source signals and computational logic.

Asset
Source Signal(s)
Computational Logic
Conceptual Meaning
blood
hr_bpm_normalized from BiophysicalEnvelopeSnapshot
Direct mapping, clamped to [0.0, 1.0]. High HR → Low blood.
Resource expenditure, cardiovascular strain.
oxygen
hrv_rmssd_normalized from BiophysicalEnvelopeSnapshot
Direct mapping, clamped to [0.0, 1.0]. High HRV → High oxygen.
Autonomic reserve, adaptability.
wave
eeg_alpha_power_norm, eeg_beta_power_norm, eeg_gamma_power_norm, eeg_alpha_cve_norm from BiophysicalEnvelopeSnapshot
Simple average of the four normalized EEG signals.
Global cortical activation and complexity.
h2o
Not Available (Placeholder)
Fixed at 0.5. Neutral placeholder until hydration axes are added.
Reserved for future hydration/metabolic proxies.
time
epoch_index from TreeOfLifeInput
Normalized linear scale over a configurable epoch window (e.g., 0-10,000 epochs).
Progression through a session.
decay
roh_score.value from TreeOfLifeInput
Normalized against the RoH ceiling of 0.3:
r
o
h
_

s
c
o
r
e
.
v
a
l
u
e
/
0.3
roh_score.value/0.3.
Proximity to the Risk-of-Harm limit.
lifeforce
roh_score.value from TreeOfLifeInput
Inverse of decay:
1.0
−
(
r
o
h
_
s
c
o
r
e
.
v
a
l
u
e
/
0.3
)
1.0−(roh_score.value/0.3). Clamped to [0.0, 1.0].
Remaining energy budget within the RoH limit.
brain
capability_state from TreeOfLifeInput
Discrete mapping: CapModelOnly(0.25) → CapGeneralUse(1.0).
Level of granted autonomy and access rights.
smart

brain score, evolve_index from TreeOfLifeInput
Blend:
0.5
×
b
r
a
i
n
+
0.5
×
n
o
r
m
a
l
i
z
e
(
e
v
o
l
v
e
_
i
n
d
e
x
)
0.5×brain+0.5×normalize(evolve_index).
Evidence-backed intelligence and capability.
evolve
evolve_index from TreeOfLifeInput
Normalized linear scale (e.g., against a max of 10,000).
Logged progression through evolutionary steps.
power
warn_axis_fraction, risk_axis_fraction from BiophysicalEnvelopeSnapshot
Weighted sum:
0.4
×

w
a
r
n
_
f
r
a
c
+
0.6
×
r
i
s
k
_
f
r
a
c
0.4×warn_frac+0.6×risk_frac.
Intensity of operation, stress level across axes.
tech
capability_state, active_axis_count from BiophysicalEnvelopeSnapshot
Blend:
0.5
×
c
a
p
a
b
i
l
i
t
y
_
s
c
o
r
e
+
0.5

×
n
o
r
m
a
l
i
z
e
(
a
x
i
s
_
c
o
u
n
t
)
0.5×capability_score+0.5×normalize(axis_count).
Overall system complexity.
fear
eda_warn_fraction, eda_risk_fraction, hr_warn_fraction, hr_risk_fraction from BiophysicalEnvelopeSnapshot
Weighted average of stressor fractions.
Sympathetic nervous system arousal (FEAR).
pain
fear score, motion_warn_fraction, motion_risk_fraction from BiophysicalEnvelopeSnapshot
Blend:
0.5
×
f
e
a
r
+
0.5
×
m
o
t
i
o

n
_
s
t
r
e
s
s
o
r
_
s
c
o
r
e
0.5×fear+0.5×motion_stressor_score.
Combined physiological and physical distress (PAIN).
nano
evolve_index from TreeOfLifeInput
Normalized linear scale (e.g., against a max of 100,000).
Granularity of logged evolutionary changes.
This comprehensive mapping demonstrates how the Tree-of-Life framework leverages the rich telemetry already available within the Neuromorphic stack to construct a multidimensional view of system state. Every calculation is transparent, traceable to a specific data source, and computationally simple, ensuring the module remains efficient and easy to audit.
Synthesis and Final Recommendations for Implementation
The research framework for implementing the Tree-of-Life as a non-actuating, educational observer layer within the NewRow-Print!/NeuroPC sovereignty stack is now fully defined. The synthesis of the provided materials reveals a coherent, robust, and secure architectural blueprint that directly addresses the user's objectives. The framework successfully navigates the delicate balance between providing rich, interpretable diagnostic insights and upholding the absolute primacy of the sovereign kernel's control invariants. It achieves this by establishing TreeofLife.rs and TreeofLife.md as the cornerstones of a system built on purity, serialization, configurability, and extensibility.
The architectural blueprint establishes a strict separation of concerns. The TreeofLife.rs module is a pure utility library, composed of data structs and side-effect-free functions, designed to transform a neuromorphic snapshot into a TreeOfLifeView . This purity is its greatest strength, guaranteeing that its outputs are deterministic and cannot, by their very nature, cause unintended state mutations or actuations. The TreeofLife.md specification serves as the constitution for this module, codifying its purpose as a read-only diagnostic and explanation surface and explicitly forbidding any deviation from this role . This dual-documentation approach ensures that both the machine-executable code and the human-understandable rules are in perfect alignment.
The framework's approach to integration is pragmatic and secure. By mandating serialization to JSON and logging into canonical data streams like .evolve.jsonl and .donutloop.aln, the Tree-of-

Life becomes an integral part of the system's official audit trail . This logging-first model decouples producers from consumers, simplifies frontend development for Unreal HUDs and AI-chat agents, and ensures that all views are based on a verifiable, immutable record of system events. This stands in stark contrast to a model of direct, real-time API calls, which would create tighter coupling and potential points of failure.

Regarding fairness diagnostics, the framework elevates the concept from a proof-of-concept to a governed practice. The initial implementation's hardcoded thresholds are identified as a limitation, and a clear path forward is prescribed: move these parameters into an external, auditable configuration file (e.g., treeoflife-config.json) that is loaded by the sovereignty core . This change transforms diagnostic sensitivity from a code-level concern to a declarative policy decision. Crucially, the framework mandates that this configuration must be accompanied by a strong governance statement in TreeofLife.md, reiterating that all diagnostic outputs are advisory labels only and must never be wired to bypass core sovereignty gates like OwnerDecision or ReversalConditions . This preserves the hierarchical integrity of the policy engine.

Finally, the framework embraces evolution through a disciplined extensibility pattern. The current implementation of placeholders for future assets like H2O is expanded into a formal protocol . This protocol, documented in TreeofLife.md, requires that any new biophysical proxy must first be formally integrated into the BiophysicalEnvelopeSpec and its governing shards before it can be consumed by the Tree-of-Life module. This ensures that the module never gets ahead of the system's validated data sources, preventing the introduction of speculative or unvetted signals. The use of Option<T> types in the Rust structs provides a robust technical mechanism for handling the conditional presence of these new signals, forcing safe handling in all downstream consumers.

Based on this comprehensive analysis, the following final recommendations are provided for the implementation of the Tree-of-Life framework:

Prioritize External Configuration: Refactor the diagnostics function in TreeofLife.rs to accept a configurable Config struct. Implement a configuration loading mechanism within the sovereignty core to read a treeoflife-config.json file on startup. Update TreeofLife.md to document these configurable parameters and to prominently feature the advisory-only disclaimer regarding their use in decision-making.

Formalize the Extension Protocol: Expand the TreeofLife.md specification to include a detailed "Future Axes and Proxies" section. This section should outline the step-by-step process for proposing, validating, and integrating a new biophysical signal, referencing the BiophysicalEnvelopeSpec and ALN governance as the governing bodies. Ensure that all new fields in the TreeOfLifeView struct are defined as Option<T>.

Enforce the Logging-First Paradigm: Document in the project's main README and developer guides that the primary method for interacting with the Tree-of-Life is by parsing the serialized logs from .evolve.jsonl and .donutloop.aln. The internal Rust API should be presented as an implementation detail for the logging mechanism itself, not as a public-facing service for ad-hoc queries.

Address Performance and Error Handling: While the current logic is computationally simple, a performance analysis should be conducted to quantify the overhead of computing the TreeOfLifeView on every neuromorphic cycle. Additionally, consider refining the error-handling strategy for missing envelope data. Instead of universally defaulting to a neutral value with .unwrap_or(), explore returning 0.0 or a specific error state for critical missing inputs, which

would make the diagnostic output more reflective of data quality issues.
By adopting these recommendations, the implementation of the Tree-of-Life will be a robust, secure, and sustainable addition to the NewRow-Print!/NeuroPC stack. It will fulfill its promise as an educational and fairness-aware diagnostic layer, providing deep insights into the system's state while respecting and reinforcing the foundational principles of neuromorphic sovereignty.

The Tree-of-Life observer layer you described is already consistent with the BiophysicalEnvelopeSpec, RoH, and capability-governance scaffolding you have in NewRow-Print!, and it can be treated as a formally specified, non-actuating diagnostic module that sits entirely on the read-only side of that stack.[27] [28] [29]

How the 14 assets fit the existing stack

Each TREE asset is grounded in an existing, non-hypothetical signal or governance object rather than new primitives:

- **BLOOD / OXYGEN**: These are just alternate visualizations of the HRBPM and HRVRMSSD-style axes already present in your BiophysicalEnvelopeSpec and cognitive-load envelope shards, with the scores derived from normalized HR and HRV values in BiophysicalEnvelopeSnapshot, then clamped into $[0.0, 1.0]$.[29] [27]
- **WAVE**: This is a composite of EEG bandpower and alpha-CVE features (eeg_alpha_power_norm, eeg_beta_power_norm, eeg_gamma_power_norm, eeg_alpha_cve_norm) that are already enumerated as DimensionKind entries and used in your cognitive-load envelope example; Tree-of-Life simply averages the normalized streams.[27] [29]
- **TIME / DECAY / LIFEFORCE / NANO**: TIME is a normalized epoch_index; DECAY and LIFEFORCE map directly to your RoH model with a hard ceiling at 0.3 (roh_score.value/0.3 and its complement, clamped to $[0.0, 1.0]$); NANO recycles evolve_index as a normalized event-count proxy, aligned with your .evolve.jsonl and .donutloop.aln append-only audit semantics.[28] [27]
- **BRAIN / SMART / EVOLVE / TECH / POWER**: These are pure functions over CapabilityState, active_axis_count, and WARN/RISK fractions that your envelope kernel already computes per axis; BRAIN reflects the discrete CAP lattice, SMART blends that with normalized evolve_index, EVOLVE exposes evolve_index directly, POWER uses WARN/RISK fractions, and TECH combines capability tier with active_axis_count as a complexity proxy.[28] [27]
- **FEAR / PAIN**: Both are weighted combinations of existing WARN/RISK fractions on EDA, HR, and motion axes, so they remain derived, advisory labels over Tier-2 envelope outputs rather than new actuation channels.[27]
- **H2O**: Explicitly fixed at 0.5 as a placeholder until hydration axes are added to BiophysicalEnvelopeSpec and governed in ALN; this matches your requirement that no speculative signal can be consumed before it exists as a validated axis.[29] [27]

Because all these mappings are clamped to $[0.0, 1.0]$ and refer only to BiophysicalEnvelopeSnapshot, TreeOfLifeInput, and already-governed CAP/ALN artifacts, Tree-of-Life can be treated as a pure view over the existing neuromorphic state space.[28] [29] [27]

Governance, non-actuation, and configuration

The role you assign to TreeofLife.rs and TreeofLife.md matches the separation of concerns already encoded in your sovereignty stack:

- TreeofLife.rs is a pure utility module (structs and side-effect-free functions) that transform a neuromorphic snapshot into TreeOfLifeView, in the same style as your envelope evaluation and CapabilityTransitionRequest::evaluate kernels.[27] [28]
- TreeofLife.md acts as a constitutional spec that explicitly forbids any actuation or capability changes, aligning with NR-SAFE-0001 and your existing doctrine "envelopes may only REQUEST, never execute, capability downgrades" and "neuromorph evolution downgrades are forbidden unless allowneuromorphreversal, explicitreversalorder, and nosaferalternative all hold."[28] [27]
- External configuration (treeoflife-config.json) and a Config struct for diagnostics thresholds and fairness parameters are fully compatible with your broader move to auditable, declarative config (e.g., .rohmodel.aln, .neuro-cap.aln, tree-level policies), and with the requirement that threshold changes become policy decisions, not code edits.[27] [28]

Because Tree-of-Life outputs are logged into .evolve.jsonl and .donutloop.aln as additional view rows but never feed into CapabilityState transitions or ReversalConditions, they remain advisory labels over an immutable audit trail, not a control surface.[28] [27]

Recommended next implementation steps

Within the organicallyintegratedaugmentedcitizen environment and Directive NR-SAFE-0001, the highest-leverage next research-object is a single, auditable Rust module that wires Tree-of-Life into your existing envelope and policy engine without creating new behavior:

1. **Refactor diagnostics into a Config-driven API**
   - Add a Config struct to TreeofLife.rs containing any tunable fairness/flag thresholds.
   - Load treeoflife-config.json in the sovereignty core alongside .rohmodel.aln and .neuro-cap.aln, and pass Config into the pure TreeofLife::from_snapshot(...) constructor.[27] [28]
   - Document the allowed parameters and advisory-only semantics in TreeofLife.md, including an explicit statement that TreeOfLifeView may never bypass OwnerDecision, CapabilityGuard, or ReversalConditions.[28]
2. **Formalize the extension protocol for new assets**
   - In TreeofLife.md add a "Future Axes and Proxies" section that requires any new biophysical proxy (e.g., hydration for H2O) to be first added to BiophysicalEnvelopeSpec and governed by ALN shard(s) before Tree-of-Life can consume it.[29] [27]
   - Enforce Option<T> for all new fields on TreeOfLifeView and require runtime checks so downstream consumers must handle absent signals explicitly, consistent with your existing Option-heavy envelope structs.[29] [27]
3. **Enforce logging-first interaction**

- In your main README and developer docs, state that the supported way to use Tree-of-Life is by reading serialized TreeOfLifeView entries from .evolve.jsonl/.donutloop.aln, mirroring how other kernels surface decisions and RoH deltas.[27] [28]
- Present any direct Rust API as an internal detail for the logging path, not as a public service, which matches your broader pattern where .evolve.jsonl is the only legal entry point and .donutloop.aln the canonical history.[28] [27]

4. **Tighten performance and error semantics**
   - Add a small profiling pass in the neuromorph policy engine to quantify the overhead of computing TreeOfLifeView on every cycle; with current logic (averages, clamps, simple blends) this should be negligible, but the measurement gives you auditable evidence.[27]
   - Replace blanket unwrap_or(neutral) usage with explicit handling: 0.0 for missing non-critical inputs and a distinct "data_missing" or "unknown" flag for critical assets; this keeps diagnostics honest about data quality and matches your stance that unknown thresholds must route into safest-first behavior rather than hidden assumptions.[28] [27]

Suggested research route for next turns

To stay within NR-SAFE-0001 and converge toward a single professional-quality implementation file, a good next research-object is:

- **ReversalConditions + CapabilityGuard integration for Tree-of-Life outputs**: specify how, if at all, any Tree-of-Life diagnostic labels (e.g., FEAR, PAIN, DECAY ≈ 1.0) can influence higher-level audit views or human decision support, while remaining strictly non-binding with respect to CapabilityState and neuromorph evolution monotonicity.[27] [28]

In a follow-up turn, you can then ask to map that spec into one Rust file (e.g., policyengine/src/tree_of_life_view.rs) that:

- Takes BiophysicalEnvelopeSnapshot, TreeOfLifeInput, and Config.
- Computes all 14 TreeOfLifeView assets in a pure, testable way.
- Emits only serialized logs for .evolve.jsonl/.donutloop.aln consumption, with no pathways to call CapabilityTransitionRequest::evaluate, ReversalConditions, or actuation code.

That file will keep Tree-of-Life firmly in its intended role: an educational, fairness-aware, read-only observer that expands interpretability without altering your neuromorphic sovereignty guarantees.[29] [28] [27]

⁂

Tree-of-Life fits cleanly as an educational, non-actuating “energy budget” and fairness layer on top of your existing NewRow-Print!/BiophysicalEnvelopeSpec stack, using BLOOD, OXYGEN, WAVE, H20, TIME, DECAY, LIFEFORCE, BRAIN, SMART, EVOLVE, POWER, TECH, FEAR, PAIN, NANO as biophysical-logic assets while still respecting RoH, neurorights, and safest-first governance.if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md+1

1. What Tree-of-Life adds
   Treats each asset (BLOOD, OXYGEN, etc.) as a scalar or index bound to existing envelope axes (HR/HRV, EDA, EEG, motion), so nothing new is “freehand” or hypothetical.[ppl-ai-file-upload.s3.amazonaws]
   Provides a didactic layer: every neuromorphic pattern is evaluated in terms of how it distributes these assets, but the only outputs are labels, scores, and logs, not actuation or rewards.newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1
   Remains deviceless and simulation-first: Tree-of-Life lives in CapModelOnly/CapLabBench, never opens a new path to CapControlledHuman beyond what the BiophysicalEnvelope and CapabilityState lattice already allow.newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1
   Asset → envelope alignment (illustrative, non-reward)
   BLOOD, OXYGEN → HR, HRV, possibly future SpO2-like streams (if present) mapped through existing RangeSpec.[ppl-ai-file-upload.s3.amazonaws]
   WAVE → EEG bandpower and alpha-envelope CVE axes you already defined.[ppl-ai-file-upload.s3.amazonaws]
   H20 → hydration or slow-acting metabolic proxies (if you later add them as axes); currently can be a neutral, “not configured” placeholder.
   TIME, DECAY → epoch counters, cooldownsecsmin, and RoH trajectory over time.[ppl-ai-file-upload.s3.amazonaws]
   LIFEFORCE → RoH global ceiling (0.30) plus neurorights shards; Tree-of-Life can only read that budget, never raise it.newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1
   BRAIN, SMART, EVOLVE → capability state and evidence bundles (balancestabilityreport, CAPMODELONLY→CAPLABBENCH→CAPCONTROLLEDHUMAN).a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
   POWER, TECH → which modules are active and at what intensity, still constrained by POLICY_STACK and .neuro-cap.aln tiers.[ppl-ai-file-upload.s3.amazonaws]
   FEAR, PAIN → combinations of EDA, HR/HRV, motion/gaze patterns that cross WARN/RISK bands in specific axes.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
   NANO → how many discrete events are written to donutloop/evolve (granularity of logged changes, not any physical nanotech).[ppl-ai-file-upload.s3.amazonaws]

2. Fairness and “no one can overpower” guarantees
   Tree-of-Life can be framed as a read-only fairness oracle that observes the policy engine; it never grants or revokes rights or capabilities.[ppl-ai-file-upload.s3.amazonaws]
   It reads CapabilityState, consent states, RoH, and envelope outputs (INFO/WARN/RISK) and computes fairness views like “no subject’s TREE assets are drained faster than others under similar conditions,” but only as metrics.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
   Any attempt to use TREE scores to auto-elevate access_grant or bypass envelopes would be blocked by the same CapabilityGuard, ReversalConditions, and .stake.aln invariants you already enforce.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1
   Unknown thresholds (e.g., new mappings from WAVE or NANO) stay in CapModelOnly with simulation-only outputs until backed by data and encoded as proper BiophysicalEnvelope axes; Tree-of-Life must treat them as “unknown, cannot decide.”a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1

3. Learning and scaling safely with AI
   Tree-of-Life can “learn” in the same way the rest of NewRow-Print! learns: via new ALN shards and Rust modules that go through .evolve.jsonl, RoH, neurorights, and CapabilityGuard.[ppl-ai-file-upload.s3.amazonaws]
   Patterns similar to existing safe ones (e.g., new combinations of BLOOD+WAVE mapped to known HR+EEG envelopes) can be explored freely in CapModelOnly with no live coupling.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
   Any proposal to let Tree-of-Life influence real-time decisions at CapControlledHuman must carry evidence bundles and stay inside BiophysicalEnvelopeSpec rohceiling and neurorights limits, and still cannot override OwnerDecision or ReversalConditions.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1
   Eco-net/evolution rewards remain out-of-scope: Tree-of-Life shards must explicitly be tagged as education-only and never feed into token, score, or selection mechanisms that could be interpreted as incentives.[ppl-ai-file-upload.s3.amazonaws]

4. Suggested research-object for next turns
   Given your description, a precise next research-object is:
   A non-actuating TreeOfLifeView ALN/Rust layer that maps envelope outputs and RoH into the 14 TREE assets, producing only educational labels, fairness diagnostics, and audit logs, with no paths into CapabilityState, consent, or rewards.
   Future turns can then focus on:
   Defining a TreeOfLifeView schema (SECTION/ROW style) that references existing BiophysicalEnvelopeSpec and RoH axes, but adds no new control.a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md+1
   Specifying fairness predicates (e.g., per-role/per-subject limits on TREE asset drift) as pure checks that can never escalate privileges.[ppl-ai-file-upload.s3.amazonaws]
   Finally, compiling that into one Rust module that plugs into the existing policy engine as an observer-only diagnostics surface, consistent with NR-SAFE-0001 and your non-fiction, executable requirements.if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md+1

Your framing is consistent with the existing NewRow-Print!/BiophysicalEnvelopeSpec and policy engine stack: Tree-of-Life fits as a pure, deviceless, educational “energy budget + fairness” observer that never touches CapabilityState, consent, or rewards.[30] [31] [32]

Role and alignment

- Tree-of-Life treats BLOOD, OXYGEN, WAVE, H2O, TIME, DECAY, LIFEFORCE, BRAIN, SMART, EVOLVE, POWER, TECH, FEAR, PAIN, NANO as scalar views over already-governed telemetry: HR/HRV, EEG and alpha-CVE, EDA, motion, epoch_index, RoH, CapabilityState, evolve_index, active_axis_count, and WARN/RISK fractions from BiophysicalEnvelopeSnapshot.[31] [30]
- All mappings are read-only and normalized to $[0.0, 1.0]$, and H2O is explicitly "neutral/not-configured" until hydration axes exist in BiophysicalEnvelopeSpec, which matches the layered envelope schema and Tier-1/Tier-2 split you already use.[30] [31]
- Tree-of-Life runs inside the same safest-first, virtual-lab-first architecture: it can operate freely in CapModelOnly/CapLabBench, and at CapControlledHuman it can only read RoH (ceiling 0.30), neurorights shards, and envelope outputs that are already constrained by .rohmodel.aln, .stake.aln, and POLICYSTACK.[32] [30]

Fairness and "no one can overpower"

- Fairness diagnostics you describe ("no subject's TREE assets are drained faster than peers under comparable conditions") can be defined as pure predicates over existing logs (CapabilityState, consent, envelope INFO/WARN/RISK, RoH trajectory), with outputs limited to labels and metrics in .evolve.jsonl/.donutloop.aln.[31] [30]
- Any attempt to drive capability elevation, access_grant, or envelope relaxation from TREE scores would still be intercepted by CapabilityTransitionRequest::evaluate, CapabilityGuard, ReversalConditions, and .stake.aln signatures, which already encode "no downgrade without OwnerDecision; no neuromorph evolution reversal unless allowneuromorphreversal, explicitreversalorder, and nosaferalternative all hold."[32] [30] [31]
- Unknown thresholds or new asset combinations must remain simulation-only: until a mapping is backed by data and encoded as a BiophysicalEnvelope axis with minsafe/maxsafe and RoH weights, Tree-of-Life should surface "unknown / not configured" rather than inventing bounds, consistent with NR-SAFE-0001.[30] [31] [32]

Learning and scaling safely

- Tree-of-Life "learns" only through new ALN shards and Rust modules that go through the same .evolve.jsonl → RoH → neurorights → CapabilityGuard pipeline as the rest of NewRow-Print!: no side-channel configuration, no implicit policy.[32] [30]
- Pattern exploration (e.g., new BLOOD+WAVE composites) can proceed indefinitely in CapModelOnly, using existing envelope ranges as reference, with no live coupling or right-bearing subjects involved.[31] [32]
- If, later, certain Tree-of-Life views are allowed to influence human-facing decisions at CapControlledHuman (e.g., raising a UI advisory when DECAY≈1.0), they must still:
  - stay within BiophysicalEnvelopeSpec rohceiling 0.30 and non-relaxing minsafe/maxsafe,
  - never bypass OwnerDecision, ReversalConditions, or composite POLICYSTACK,

- and be explicitly tagged “education-only, non-reward, non-incentive” in their ALN shards so they cannot be repurposed as eco-net or evolution reward signals.[30] [31] [32]

Next research-object

The next precise object you sketched is well-posed:

- Define a TreeOfLifeView schema in SECTION/ROW style that references existing BiophysicalEnvelopeSpec axes, RoH, CapabilityState, evolve_index, and envelope WARN/RISK fractions, but provides only:
  - the 14 TREE asset scores,
  - optional fairness diagnostics,
  - and immutable logs for .evolve.jsonl/.donutloop.aln.[31] [30]
- Then implement a single Rust module (e.g., policyengine/src/tree_of_life_view.rs) that:
  - takes BiophysicalEnvelopeSnapshot + TreeOfLifeInput,
  - computes the 14 assets as pure functions,
  - writes TreeOfLifeView records into the canonical log streams,
  - exposes no API to modify CapabilityState, consent, RoH models, or rewards.[32] [30]

If you want, the next turn can focus just on that schema: we can draft the TreeOfLifeView SECTION/ROW surface (META, INPUTS, ASSETS, FAIRNESS, OUTPUTS) so it is code-generatable and plugs cleanly into your existing envelope and policy engine without adding any new control paths.[30] [31] [32]

⁂

We can treat that as accepted and now define the role explicitly in both ALN and Rust terms, so it is always concrete and auditable.

newrow-print-l_myn4yfSA6t9spUFtJA4w.md+1

ALN side: NEUROMORPH-GOD alias role
Define the composite role once in a governance shard, as a pure alias:
text
SECTION,ROLES
ROW,role,base,enum,HOST,Host,string,nonnull,Primary host role
ROW,role,base,enum,ORGANICCPU_OWNER,OrganicCPUOwner,string,nonnull,Owner of OrganicCPU
ROW,role,base,enum,REGULATOR,Regulator,string,nonnull,Regulatory or ethics authority
ROW,role,base,enum,SOVEREIGN_KERNEL,SovereignKernel,string,nonnull,Internal kernel role

ROW,role,alias,enum,NEUROMORPH-GOD,NeuromorphSovereign,string,nonnull,
Host + OrganicCPUOwner + Regulator quorum + SovereignKernel

SECTION,ROLE-COMPOSITION
ROW,rc,neuromorph,condition,neuromorphgod_satisfied,
hasroleHost AND hasroleOrganicCPUOwner
AND hasquorumRegulator >= 2
AND hassystemroleSovereignKernel,
string,readonly,Composite role predicate

FOOTER,END-OF-SHARD

Interpretation: any ALN rule that requires NEUROMORPH-GOD simply expands to the concrete predicate neuromorphgod_satisfied.[ppl-ai-file-upload.s3.amazonaws]
You can then replace earlier narrative constraints with a single guard, for example:
text
ROW,rev,capability,condition,canrevertcapability,
neuromorphgod_satisfied AND explicitreversalorder true AND nosaferalternative true,
string,readonly,Reversal last-resort condition

This keeps ReversalConditions exactly as you specified, but pins them to the new composite role.[ppl-ai-file-upload.s3.amazonaws]
Rust side: Role::NeuromorphSovereign
Extend the existing Role enum and add a helper:
rust
// filename: policyengine/src/aln_roles.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
pub enum Role {
Mentor,
Teacher,
Learner,
Operator,
Regulator,
System,
Host,
OrganicCpuOwner,
SovereignKernel,
NeuromorphSovereign, // NEUROMORPH-GOD alias
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub struct RoleSet {
pub roles: Vec<Role>,
pub regulator_quorum: u8,
}

impl RoleSet {
pub fn has(&self, r: Role) -> bool {
self.roles.contains(&r)
}

```
/// “Neuromorph-god” composite predicate:
/// Host + OrganicCPUOwner + SovereignKernel + regulator quorum.
pub fn neuromorph_god_satisfied(&self, required_regulator_quorum: u8) -> bool {
    self.has(Role::Host)
        && self.has(Role::OrganicCpuOwner)
        && self.has(Role::SovereignKernel)
        && self.regulator_quorum >= required_regulator_quorum
}
```

}

Kernel code that enforces ReversalConditions can now depend on this single predicate instead of repeating the composition:
rust
pub fn can_revert_capability(
roles: &RoleSet,
required_reg_quorum: u8,
explicit_reversal_order: bool,
no_safer_alternative: bool,
) -> bool {
roles.neuromorph_god_satisfied(required_reg_quorum)
&& explicit_reversal_order
&& no_safer_alternative
}

This keeps your mythic label as a strictly symbolic alias for a concrete, multi-sig, multi-role sovereignty condition, fully aligned with NR-SAFE-0001 and your reversal monotonicity guarantees.newrow-print-I_myn4yfSA6t9spUFtJA4w.md+1

Your ALN and Rust sketches are consistent with the existing sovereignty, .stake.aln, and ReversalConditions model and can be adopted essentially as-is, with only minor tightening to keep everything explicit and auditable.[33] [34] [35]

ALN: NEUROMORPH-GOD alias role

Your ROLE and ROLE-COMPOSITION shard is structurally sound and lines up with the Tier-1 foundation and reversal policy sections already defined.[34] [33]

```
SECTION,ROLES
ROW,role,base,enum,HOST,Host,string,nonnull,Primary host role
ROW,role,base,enum,ORGANICCPU_OWNER,OrganicCPUOwner,string,nonnull,Owner of OrganicCPU
ROW,role,base,enum,REGULATOR,Regulator,string,nonnull,Regulatory or ethics authority
ROW,role,base,enum,SOVEREIGN_KERNEL,SovereignKernel,string,nonnull,Internal kernel role

ROW,role,alias,enum,NEUROMORPH-GOD,NeuromorphSovereign,string,nonnull,
 Host + OrganicCPUOwner + Regulator quorum + SovereignKernel

SECTION,ROLE-COMPOSITION
ROW,rc,neuromorph,condition,neuromorphgod_satisfied,
 hasroleHost AND hasroleOrganicCPUOwner
```

```
 AND hasquorumRegulator >= 2
 AND hassystemroleSovereignKernel,
 string,readonly,Composite role predicate

SECTION,REVERSAL-POLICY
ROW,rev,capability,flag,allowneuromorphreversal,false,bool,nonwaivable,No evolution rever
ROW,rev,capability,flag,explicitreversalorder,,bool,input,Owner-signed reversal order
ROW,rev,capability,flag,nosaferalternative,,bool,derived,All soft mitigations exhausted
ROW,rev,capability,condition,canrevertcapability,
 neuromorphgod_satisfied AND explicitreversalorder true AND nosaferalternative true,
 string,readonly,Reversal last-resort condition

FOOTER,END-OF-SHARD
```

This preserves the existing invariant (“no neuromorph evolution reversal unless explicit owner order and no safer alternative”) and simply pins it to a composite, multi-sig, multi-role predicate, in line with your Tier-1 foundation shard and neurorights framing.[33] [34]

Rust: Role::NeuromorphSovereign and helper

Extending the Role enum and introducing RoleSet as you proposed matches the existing alncore.rs capability/role model and keeps the composite condition auditable in one place.[35]

```
// filename: policyengine/src/aln_roles.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
pub enum Role {
    Mentor,
    Teacher,
    Learner,
    Operator,
    Regulator,
    System,
    Host,
    OrganicCpuOwner,
    SovereignKernel,
    NeuromorphSovereign, // NEUROMORPH-GOD alias (symbolic)
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub struct RoleSet {
    pub roles: Vec<Role>,
    /// Number of independent Regulator signatures observed for this decision.
    pub regulator_quorum: u8,
}

impl RoleSet {
    pub fn has(&self, r: Role) -> bool {
        self.roles.contains(&r)
    }

    /// “Neuromorph-god” composite predicate:
    /// Host + OrganicCPUOwner + SovereignKernel + regulator quorum.
    pub fn neuromorph_god_satisfied(&self, required_reg_quorum: u8) -> bool {
```

```
        self.has(Role::Host)
            && self.has(Role::OrganicCpuOwner)
            && self.has(Role::SovereignKernel)
            && self.regulator_quorum >= required_reg_quorum
    }
}

/// Kernel helper for ReversalConditions.
pub fn can_revert_capability(
    roles: &RoleSet,
    required_reg_quorum: u8,
    explicit_reversal_order: bool,
    no_safer_alternative: bool,
) -> bool {
    roles.neuromorph_god_satisfied(required_reg_quorum)
        && explicit_reversal_order
        && no_safer_alternative
}
```

In the ReversalConditions kernel, you can now treat any neuromorph evolution downgrade as:

- forbidden by default (allowneuromorphreversal = false in Tier-1), and
- only allowed along the single, narrow path where:
  - allowneuromorphreversal is true for that capability,
  - can_revert_capability(...) returns true, and
  - the usual CapabilityTransitionRequest, .stake.aln, POLICYSTACK, and RoH checks all pass.[34] [35] [33]

This keeps **NeuromorphSovereign** as a purely symbolic alias for a concrete, multi-party sovereignty condition, fully consistent with NR-SAFE-0001 and your monotone-evolution guarantees.[35] [33] [34]

⁂

1. newrow-print-I_myn4yfSA6t9spUFtJA4w.md
2. if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md
3. a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md
4. newrow-print-I_myn4yfSA6t9spUFtJA4w.md
5. if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md
6. a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md
7. if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md
8. newrow-print-I_myn4yfSA6t9spUFtJA4w.md
9. a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md
10. if-necessary-sanitize-the-code-7jDmbRJIT3SnSttCB78ZQg.md
11. newrow-print-I_myn4yfSA6t9spUFtJA4w.md
12. a-foundational-report-on-neura-kVDayD8JSie4.ukpYI37WA.md
13. newrow-print-I_myn4yfSA6t9spUFtJA4w.md

14. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
15. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
16. create-a-heavy-research-plan-f-iuQRhxq3SXKEqzqKASlSog.md
17. neuropcs-rules-and-goals-are-c-bJlTjTqfQHaJgTu_2pFVnw.md
18. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
19. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
20. organic-solutions.biosafe.txt
21. Galea-Varjo-Aero.txt
22. EEG-Bandpower.pdf
23. hybrid-bootloader-ai-constrain-lOh.MQwYSTqwCXUuWL7L3g.md
24. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
25. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
26. newrow-print-l_myn4yfSA6t9spUFtJA4w.md
27. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
28. newrow-print-l_myn4yfSA6t9spUFtJA4w.md
29. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
30. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
31. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
32. newrow-print-l_myn4yfSA6t9spUFtJA4w.md
33. a-foundational-report-on-neura-kVDayD8JSie4.ukpYl37WA.md
34. if-necessary-sanitize-the-code-7jDmbRJlT3SnSttCB78ZQg.md
35. newrow-print-l_myn4yfSA6t9spUFtJA4w.md