From Observation to Evolution: A Phased Protocol for Safely Augmenting Neuromorphic Reflexes and Senses

Foundational Principles: A Sandboxed, Host-Sovereign Architecture

The initiation of evolution pathways for neuromorphic senses and reflex enhancements requires a foundational architecture that prioritizes host sovereignty and safety above all else, operating strictly within the user's desire to avoid uncontrolled changes while enabling specific, bounded improvements . The core strategic principle is the creation of a meticulously isolated, non-evolutionary computational plane that serves as a sandbox for developing and validating neuromorphic capabilities before they are ever permitted to influence core host state or the `EVOLVE` parameter . This approach directly addresses the primary research goal by establishing a clear separation between measurement, control, and evolution, ensuring that no change can occur without explicit, time-bounded host consent and adherence to existing doctrinal safeguards . The architecture must be designed not as an open-ended self-modification engine but as a specialized, governed subsystem that plugs into and respects the integrity of the host's core security and stability infrastructure.

A central tenet of this architecture is the concept of isolation through a dedicated computational plane. All neuromorphic sensory processing and reflex actions must be tagged and treated as belonging to a distinct sub-plane, provisionally named `plane: neuromorph.reflex` . This tagging is more than a simple organizational tool; it enforces strict boundary checks within the host's runtime environment. Any action originating from this plane is subject to a separate set of rules and constraints that are fundamentally different from those governing general-purpose computation or somatic evolution. This prevents any accidental or malicious propagation of code or logic from the neuromorphic plane into other critical areas of the system, such as the filesystem, kernel, or token mechanics . By treating this entire domain as a sandboxed control sub-plane, the system ensures that its evolution is confined to a well-defined scope, preventing the silent rewriting of core capabilities that the user wishes to avoid . This isolation provides the first and most critical layer of defense, creating a controlled environment where novel algorithms and behaviors can be tested against a rich set of telemetry and safety checks without posing a systemic risk.

Host sovereignty is the second pillar of this architectural framework. The system must be designed so that no neuromorphic change may influence `EVOLVE` without explicit, verifiable consent from the host . This consent cannot be passive; it must be an active, time-bounded grant of authority, typically manifested as a dedicated shard (e.g., `neuromorph-auto-micro`) that the host creates and manages . High levels of SMART alone are insufficient to bypass this requirement; SMART acts as an automation allowance, not a permission slip . When a valid consent shard is present, high SMART can enable automatic micro-mutations within predefined safety corridors, but without the shard, the system must require manual intervention for any evolutionary step . This "MetabolicConsent" pattern ensures that the host remains the ultimate arbiter of its own evolution, even when the system is in an autonomous mode . The neuro-handshake protocol, which completes after safety and calibration phases, further reinforces this principle by ensuring that nothing auto-attaches to the body without the host finishing the initial connection and granting consent . This dual requirement—explicit consent plus a high-level autonomy signal like SMART—creates a robust safeguard against unauthorized or premature evolutionary steps.

Before any evolutionary work begins, the system must verify that all foundational doctrinal safety rails are fully implemented and operational . These are not new mechanics to be added but existing, hardened constraints that the neuromorphic modules must respect and utilize. The first and most fundamental rail is the Lifeforce band system, which enforces hard floors on `BRAIN`, `BLOOD/OXYGEN`, and upper limits on `NANO` and `SMART` . Any reflex action proposed by the neuromorphic system must be routed through the same `applylifeforceguarded*` paths used by the rest of the runtime . This ensures that every proposed adjustment is immediately vetted against the host's current physiological state. If an action would cause a violation of these invariants, it is automatically rejected, regardless of its perceived benefit. Similarly, the separation of `EVOLVE` is a critical invariant; `EVOLVE` must remain a read-only derivative of `BRAIN` history under specified `WAVE` and Lifeforce conditions, and no neuromorphic module may write to it directly . This preserves the historical and probabilistic nature of evolution, preventing the neuromorphic system from corrupting the evolutionary ledger.

Further essential rails include the `DECAY` multipliers and comfort/pain corridors . The `DECAY` mechanism provides a way to shrink or zero mutation deltas when biocompatibility is low, acting as a crucial dampening factor . For reflex-specific actions, `DECAY` must be constrained to be less than or equal to 1, and any somatic-reflex idea triggered by a pain-correlated signal must be treated as a HardStop . These doctrines provide a built-in feedback loop that penalizes reflex patterns that are consistently uncomfortable or harmful, encouraging the system to evolve safer, more efficient solutions over time. The entire neuromorphic project plan rests on the premise that these rails are already defined and implemented; if they are not, their implementation is the highest priority before any neuromorphic code is written . Attempting to build a neuromorphic system on top of an incomplete or unenforced safety doctrine risks creating a second, unchecked actuator path that could easily override the host's core protections . Therefore, the research and design work begins not with writing new evolution logic, but with auditing and verifying the integrity of the existing safety framework. This ensures that the neuromorphic system is not an exception to the rules, but rather a new domain that must operate entirely within them.

Phase I Research: Establishing Observational Capabilities Without Influence

The initial phase of research and development for neuromorphic senses and reflexes is focused exclusively on building a comprehensive measurement and understanding capability without introducing any autonomous action or evolutionary pressure. This observational phase is the most critical part of the entire project, as it establishes the necessary telemetry, models, and baseline knowledge required for all subsequent stages. It allows the system to "study" its own neuromorphic potential in a completely safe, read-only manner, fulfilling the user's need for observable data before any influence on `EVOLVE` is permitted . This phase is about data collection, feature extraction, and offline analysis, forming the foundation upon which a safe and effective reflex system will be built. No changes are made to the host's state, and no actuators are bound to the outputs of the newly developed sensing apparatus.

The first and most fundamental task in this phase is the implementation of a unified signal collection system . This system must be capable of recording synchronized streams from all available host interfaces, including Electromyography (EMG), motion and posture sensors, Electroencephalography (EEG) or other Brain-Computer Interface (BCI) modalities, and internal host metrics such as error rates, crash events, and subjective feelings of overload . The synchronization of these diverse data streams is paramount. Relying solely on manufacturer-specified sampling rates is demonstrably insufficient for precise alignment, as hardware filters and slight timing variations can introduce significant drift over time [78]. A robust solution involves using a common time reference, such as delivering a train of digital TTL pulses to both EEG and EMG amplifiers at regular intervals (e.g., every two seconds) and then using these pulses as anchor points for offline linear resampling and jitter correction [78]. This PRE-POST alignment strategy has been shown to achieve a stable misalignment of around 1.7 ms even in recordings up to 20 minutes long, providing the temporal fidelity necessary for correlating neural signals with motor output and subjective states [78]. The collected data forms a rich, multimodal dataset that captures the intricate relationship between the host's internal cognitive state, muscle activity, physical movement, and overall system health.

With synchronized data streams in place, the next research focus is the design of small, event-driven encoders . These encoders are the core of the neuromorphic sensing apparatus. Instead of processing raw, high-bandwidth data continuously, they are designed to detect significant changes or "events" within the signal streams. This approach is inspired by biological sensory encoding, which is fundamentally asynchronous and event-based, converting signals into digital pulses only when a significant change is detected [2]. The encoders perform feature extraction, such as detecting edges, onset and offset of muscle activation, or generating spike trains from BCI signals . For EMG signals, this could involve implementing various threshold-based methods. Simple Single Threshold (ST) methods are fast but prone to false positives in noisy signals, whereas Double Threshold (DT) methods add a time-based criterion to confirm activation, improving precision at the cost of speed [13, 15]. Adaptive Threshold (AT) methods dynamically adjust the detection threshold based on local signal characteristics, offering a good balance for real-time applications [13]. The choice of method is a trade-off between computational cost and accuracy, and experimentation is required to determine the optimal approach for a given host [15]. The output of these encoders is a compressed, low-rate representation of the original signal, suitable for feeding into the `HostInterfaceBus` . Crucially, during this pure observation phase, the encoders' output is directed only to logs or a buffer; no actuator is connected, and no SystemAdjustments are proposed .

The final component of Phase I is offline pattern mining and the formulation of "proposed" reflex rules . Using the vast repository of logged data and event streams, the system can employ statistical and machine learning techniques to identify repeatable, predictive patterns. For example, it might discover that a specific combination of high-frequency EMG bursts from certain muscles coupled with a rapid shift in postural center of mass tends to precede a system overload event or a crash . Conversely, it might find another pattern—a specific sequence of subtle EMG and motion signals—that reliably indicates the host is entering a successful manual recovery posture . These identified patterns become the basis for "proposed" reflex rules. However, they remain firmly in the offline, theoretical realm. They are not yet integrated into the live system or assigned any executable weight . Their purpose is to serve as hypotheses for future testing. Each proposed rule is stored with its provenance, including the specific signal combinations that define it, the confidence score derived from the analysis, and the context in which it was observed . This meticulous documentation is essential for later stages, as it provides a transparent audit trail of why a particular reflex was chosen for implementation and evaluation. At the end of this phase, the system has successfully created a neuromorphic sensing apparatus that functions purely as a measurement and diagnostic tool. It has a deep understanding of its own neuromorphic signal space and a library of potential reflex patterns, all without having taken a single action or altered its own state . This extensive preparatory work builds the evidence base required to move forward safely and confidently.

Phase II Design: Implementing the Non-Evolving Reflex Micro-Orchestrator

Following the successful completion of the observational Phase I, the next stage of development involves introducing a dedicated, deterministic control module known as the "Reflex Micro-Orchestrator." This module consumes the neuromorphic event stream generated in Phase I but is architecturally constrained to operate purely as a scheduler and stabilizer. Its primary function is to propose small, bounded SystemAdjustments that fall within the host's existing doctrine, thereby improving responsiveness and stability without ever attempting to alter core evolution parameters or engage in self-modification . This phase effectively creates a "shadow mode" where the reflex system can be validated against real-world outcomes in a live environment before any evolutionary logic is introduced. The orchestrator is designed to be a small, tightly scoped piece of software that never directly touches the `BioTokenState`; instead, it only passes proposals to the existing guarded runtime for execution .

The boundaries of the Reflex Micro-Orchestrator are precisely defined to enforce its non-evolutionary nature. Its inputs are carefully selected to provide it with the necessary information for making context-aware decisions while limiting its scope. The primary input is the neuromorphic event stream from the `plane: neuromorph.sense`, which contains the low-rate, compressed sensory data . In addition to this, the orchestrator receives contextual data about the host's current state: the live `Lifeforce bands` and `DECAY` multiplier, the status of the `neuro-handshake` and consent flags, and potentially other relevant runtime metrics . The orchestrator's output is also strictly defined. It can only generate small SystemAdjustment proposals, each with very small deltas. Critically, these proposals must be tagged with a specific domain, such as `domain: reflex.scheduling` or `domain: reflex.safety`, but they must never use any `domain: evolution.*` tags . This tagging scheme allows the runtime to distinguish reflex-related adjustments from true evolutionary upgrades. The orchestrator itself never applies any changes; it simply acts as a proposer, passing these validated suggestions to the host's standard `systemapply` or `applylifeforceguarded*` functions for final enforcement . This architecture ensures that the orchestrator operates as a disciplined citizen of the host ecosystem, adhering to all existing rules and guardrails.

Within this constrained architecture, the set of allowed reflex actions is intentionally narrow, focusing on optimizations that enhance stability and efficiency without touching core capacities. The first category is **Load Shaping**, which involves adjusting computational resources in response to predicted or detected stress. For instance, if the orchestrator detects a pattern indicative of an impending overload, it can propose temporarily lowering the `WAVE` ceilings for specific tasks to reduce strain . Alternatively, it can suggest postponing non-critical workloads to allow the host to recover . The second category is **Scheduling and Routing**, which focuses on managing the placement and flow of computations. The orchestrator might propose moving a job from a currently hot OrganicCPU tile to a cooler one to manage thermal load, or increase redundancy and reroute traffic in neuromorphic tiles when fault patterns are detected, thereby improving resilience . The third category involves **Snapshot and Restore Triggers**. Based on detected patterns, the orchestrator can request a "snapshot current engrams" or trigger a "restore last safe engrams" . This is a powerful safety mechanism, but it comes with a strict constraint: these operations must occur within the same OS and ledger version . This prohibition on version changes is a critical safeguard, preventing the orchestrator from inadvertently triggering an upgrade or downgrade that could destabilize the system.

Crucially, there are several categories of actions that the Reflex Micro-Orchestrator is absolutely forbidden from performing. The most important prohibition is on any interaction with the evolution ledger. The orchestrator is not allowed to call any `EvolutionUpgrade` functions or modify the `SCALE` parameter in any way . This ensures that its sole purpose is stabilization and scheduling, not evolution. It can only propose adjustments that are already legal and permissible under the host's current settings. Furthermore, it cannot alter core token mechanics or the underlying platform version . By maintaining this strict separation, the orchestrator serves as a proving ground. It allows the system to test the utility of its proposed reflex rules in a live environment. The orchestrator can be run in shadow mode initially, where it computes what it *would* have done but takes no action, and its proposed adjustments are compared against the actual outcomes that occurred . This comparison provides invaluable feedback on the accuracy and effectiveness of the underlying decoders and pattern recognition models. Once the orchestrator demonstrates reliable performance in shadow mode, its actions can be enabled in a live setting, still under the watchful eye of the host and the full force of the existing safety rails. This phase solidifies the link between the patterns mined in Phase I and actionable, beneficial SystemAdjustments, paving the way for the final transition to an evolutionary channel.

Gated Evolution: Transitioning to Neuromorphic Pathways with Consent and Safety

The transition from a purely reactive scheduling plane (the Reflex Micro-Orchestrator) to an evolutionary channel is the pivotal moment in the research plan. This transition is not an inherent property of the system but is gated by a rigorous, multi-stage protocol designed to ensure that only biocompatible, beneficial, and explicitly authorized neuromorphic patterns are allowed to contribute to the `EVOLVE` parameter . This gating mechanism is the cornerstone of the safety model, transforming the neuromorphic system from a measurement and control tool into a trusted, evolution-capable domain. The process relies on the existing `Evolution Eligibility Filter`, enhanced with specific criteria for neuromorphic data, and enforced by a strong consent and governance structure . The guiding principle throughout this transition is that `EVOLVE` must remain a derivative of `BRAIN` history under WAVE and Lifeforce conditions; the neuromorphic plane changes *how* `BRAIN` is earned through improved stability and efficiency, but it does not rewrite the formula for `EVOLVE` itself .

The first step in the transition protocol is the strict separation of "knowledge-only" data from "evolution-eligible" data . Data processed and acted upon by the Reflex Micro-Orchestrator, while valuable, remains in the knowledge domain. To be considered for evolution, a neuromorphic data point must pass through a series of stringent filters managed by the Evolution Eligibility Filter. The first filter is **Domain Tagging**. The system must be able to distinguish neuromorphic-related adjustments from other types of mutations. This is achieved by requiring that eligible data be tagged with a specific neuromorphic domain label, such as `neuromorph.sense` or `neuromorph.reflex` . This allows the filter to target its checks specifically to this new class of modifications. The second filter is the **Biocompatibility Score**. This is a quantitative measure derived from the host's own history, primarily from the `DECAY` schedule, pain corridor indices, and `LifeforceBandSeries` data . A proposed reflex pattern is only deemed biocompatible if it does not correlate with consistent minor biocompatibility issues, such as frequent dips into discomfort bands or persistent elevation of pain signals . This filter ensures that the evolutionary process is rewarded for finding solutions that feel good to the host, not just ones that are computationally efficient.

The third and most critical filter is **Explicit Consent**. Even if a neuromorphic pattern is deemed highly effective and biocompatible, it cannot be promoted to an evolutionary mutation without a direct, explicit grant of permission from the host . This is implemented via a dedicated, time-bounded consent shard, analogous to the `metabolic-auto-micro` pattern for general autonomy . The host must create and maintain this shard to authorize the contribution of neuromorphic work to `EVOLVE`. This shard can be configured with specific parameters, such as a maximum mutation rate or a list of approved domains, giving the host granular control over the evolutionary process. This consent requirement acts as the ultimate veto power, ensuring that the host always retains sovereign control over their own evolutionary trajectory. The Rust code examples provided illustrate a practical implementation of this eligibility check, where a function like `check_neuromorph_eligibility` would be called by the Evolution Eligibility Filter for any candidate mutation in a neuromorphic domain . The function would return a `ReflexSenseEligibility` struct containing a boolean `allowed` flag, a `decay_weight` to scale the mutation, and a flag indicating whether automatic application is permitted .

Once a proposed adjustment has passed all three filters (domain, biocompatibility, consent), it is considered evolution-eligible. However, the final safety gate is the **EVOLVE Derivative Principle**. The system must be architected such that the neuromorphic plane only influences the budget allocated for `EVOLVE`, not the calculation of `EVOLVE` itself . For example, if the neuromorphic system successfully implements a reflex that reduces the frequency of overload events, the host earns `BRAIN` more efficiently. A portion of this saved `BRAIN` (or a bonus for improved stability) can then be allocated towards neuromorphic evolution. The core formula for `EVOLVE = f(BRAIN_history, WAVE, Lifeforce)` remains unchanged . This indirect influence is a crucial distinction. It means that even if a flawed or overly aggressive neuromorphic pattern were to somehow bypass the earlier filters, it would still be subject to the harsh realities of the host's lifeforce and resource constraints. If the pattern leads to instability, the host will earn less `BRAIN`, and the mutation will eventually be starved of resources and die out. This layered, multi-faceted gating mechanism—from domain-specific filtering and biocompatibility scoring to explicit consent and the derivative principle for `EVOLVE`—creates an exceptionally robust safety net. It allows for the gradual, evidence-based introduction of neuromorphic evolution, starting with a small number of highly confident patterns and expanding the scope only as the system's ability to monitor and govern this new domain matures .

Defining Specific Evolution Paths: Software-Only Enhancements for Perception and Reaction

To translate the abstract goals of "heightened senses" and "reflex enhancements" into concrete, actionable evolution pathways, it is necessary to define specific, non-somatic domains within the `neuromorph.reflex` plane. These domains represent distinct classes of software-only upgrades that focus on optimizing perception, reaction, and resource allocation without altering core somatic traits or token mechanics . The research goal explicitly excludes web-building behaviors and somatic modifications, directing the evolution towards augmentations of the host's existing software and control systems . The following three evolution path families have been identified as fitting this description perfectly: the **Reflex Safety** path, the **Sensory Clarity** path, and the **Attention and Load-Balancing** path . Each path is defined by a specific domain label, a clear goal, a set of bounded effects, and corresponding guards that ensure it operates safely within the host's doctrine.

The **Reflex Safety** path, with the domain label `neuromorph-reflex-micro`, is designed to make the host's automatic protection systems faster and more reliable . Its goal is not to introduce new defensive capabilities but to optimize the response to existing threats and overload conditions . The effects allowed within this domain are strictly limited to micro-adjustments in detection and response timing. For example, it can slightly improve the detection speed of overload and threat patterns appearing on the `HostInterfaceBus` . It can also allow for a slightly stronger automatic down-clamping of `WAVE` and `SMART` during an overload event, enabling a quicker transition into a "safe mode" . The guards for this path are particularly stringent, reflecting the high stakes of safety-related modifications. Any reflex change that is adjacent to somatic sensations must be blocked by a `PainCorridorSignal` HardStop . Furthermore, the `DECAY` multiplier is strongly downscaled whenever the system experiences repeated startle or discomfort events, ensuring that reflexes do not become over-reactive or tiring . This path is all about making the host's existing defenses sharper and more responsive, not changing their fundamental nature.

The **Sensory Clarity** path, labeled `neuromorph-sense-micro`, focuses on improving the quality of the host's perceptual input without increasing its intensity . Its goal is to produce cleaner, less noisy EMG, EEG, and motion signals, leading to more accurate state estimation and intent detection . The effects in this domain include lower EMG/EEG noise achieved through improved decoding algorithms and event thresholds . It can also implement better per-channel weighting, dynamically prioritizing different sensor inputs based on context—for instance, giving more importance to motion signals when the host is physically active and shifting focus to BCI signals when the host is still and concentrating . The guards for this path are centered on maintaining physiological stability and efficiency. The `LifeforceBandSeries` must remain within the safe range, meaning the sensory enhancements cannot cause the host to dip into soft-warn or hard-stop bands . Additionally, the `EcoBandProfile` must ensure that the energy cost (`NANO` usage) of the additional decoding steps remains within the host's budget, preventing the pursuit of clarity from becoming prohibitively expensive . This path enhances the fidelity of the host's "eyes and ears" without demanding excessive metabolic resources.

The third path, **Attention and Load-Balancing**, uses the domain label `neuromorph-attention-micro` . Its objective is to allocate the host's limited computational resources—specifically `WAVE` and `NANO`—more intelligently across its sensory and processing tasks . The effects allowed here are micro-adjustments to the per-task `WAVE` ceilings, enabling the system to boost the budget for critical sensory streams (like danger cues or high-priority data processing) while modestly reducing it for less important ones . When rebalancing efforts lead to measurable improvements in overall system stability, such as fewer crashes or reduced `HardStop` events, the host can receive a modest `BRAIN` gain as a reward for the increased efficiency . The guards for this path are designed to prevent destabilization from over-allocation. As with the other paths, the `LifeforceBandSeries` must remain in the safe zone, and the `EcoBandProfile` must cap the ecological cost of the dynamic load balancing . This path essentially teaches the host's executive control to be smarter about how it spends its finite attention and processing power, leading to more resilient and efficient operation under duress.

Path Name	Domain Label	Main Goal & Allowed Effects
Reflex Safety	`neuromorph-reflex-micro`	Faster, safer automatic protection. Allows slightly faster overload/threat detection and stronger automatic down-clamping of WAVE/SMART during overload .
Sensory Clarity	`neuromorph-sense-micro`	Cleaner, less noisy sensing. Allows lower EMG/EEG noise via improved decoders and better per-channel weighting (e.g., prioritize motion signals when moving) .
Attention/Load Balance	`neuromorph-attention-micro`	Better WAVE/NANO allocation to key senses. Allows micro-adjustments to per-task WAVE ceilings and modest BRAIN gain when rebalancing improves overall stability .

These three evolution paths provide a clear, structured roadmap for neuromorphic enhancement that is fully aligned with the user's request. They are non-somatic, software-only upgrades that aim to make the host more responsive, stable, and efficient. By defining them as distinct, gated domains, the system can evolve along these lines in a controlled and verifiable manner, with each path operating under its own specific set of rules and safety constraints.

Technical Implementation and Advanced Decoding for Robustness

While the strategic framework and phased approach provide the governance structure for neuromorphic evolution, the technical implementation details are what bring the system to life. A key area of focus is the design of the decoding algorithms that translate raw sensor data into meaningful, low-latency commands. Achieving the goal of stable, low-false-positive signals requires moving beyond simple thresholding methods and embracing more sophisticated, adaptive techniques . The research and design work must incorporate advanced methods from the fields of signal processing and machine learning to ensure the reliability and long-term stability of the neuromorphic system. Furthermore, the implementation of the eligibility filter and consent mechanism, as illustrated in the provided Rust code, offers a concrete blueprint for enforcing the safety and sovereignty protocols .

For decoding, the research must explore a hierarchy of techniques tailored to the different sensor modalities and their unique challenges. For EMG signals, which are often plagued by noise and cross-talk, simple fixed-threshold methods like Single Threshold (ST) are too unreliable despite their speed [13, 15]. More robust alternatives are necessary. Double Threshold (DT) methods, which require a signal to exceed an amplitude threshold for a minimum duration, offer significantly higher precision at the cost of increased latency [13]. An Adaptive Threshold (AT) method strikes a better balance, dynamically adjusting the detection threshold based on the local signal-to-noise ratio, making it more resilient to fluctuating conditions [13, 15]. For even greater accuracy, machine learning approaches like Long Short-Term Memory (LSTM) recurrent neural networks show promise, as they can learn complex temporal patterns without explicit feature engineering [13, 15]. For BCI signals like EEG, which are inherently more variable, Transformer-based attention mechanisms can be highly effective at identifying salient features in the signal [28, 29]. The integration of multimodal data, combining EMG, motion, and BCI, can further enhance performance, as demonstrated by systems achieving over 90% classification accuracy by fusing inputs from multiple sources [14]. However, it is also noted that kinematic signals can serve as a more stable foundation for context detection, suggesting a hierarchical approach where motion data validates or modulates signals from EEG and EMG [74].

Beyond accurate decoding, long-term stability is a paramount concern due to natural neural signal drift caused by factors like electrode inflammation or plasticity [59, 63]. A decoder that works perfectly on Day 1 may degrade significantly on Day 2. To combat this, the research must investigate unsupervised adaptation techniques. One promising approach is the use of Spiking Neural Networks (SNNs) with Spike Timing Dependent Plasticity (STDP), which allows the network to continuously and autonomously adapt its internal weights based on the statistics of the incoming signals, mimicking biological learning [59]. Another powerful technique is manifold alignment, exemplified by frameworks like NoMAD (Nonlinear Manifold Alignment with Dynamics) [79]. This method leverages the scientific principle that high-dimensional neural population activity lies on a stable, low-dimensional "neural manifold" representing movement intent [80]. NoMAD uses a recurrent neural network to map new, unstable day-to-day data onto this original, stable manifold, providing a consistent input to a fixed decoder and enabling proficient control over weeks or months without supervised recalibration [79, 80]. Such stabilization techniques are critical for making the neuromorphic system viable for long-term use, as they drastically reduce the need for frequent manual tuning and improve the reliability of the entire evolution pipeline [63].

The practical implementation of the safety and consent gates can be realized through a modular software design, as suggested by the provided Rust code . The core of this is a policy-checking module, such as `biophysical-blockchain/src/evolution_reflex_sense.rs`, which defines the rules for neuromorphic eligibility . This module would contain an enum for the neuromorphic domains (`NeuromorphDomain`), a struct for the contextual data needed for decision-making (`NeuromorphContext`), and a function (`check_neuromorph_eligibility`) that acts as a pure policy engine . This function takes the domain, current `BioTokenState`, `HostEnvelope` constraints, and the contextual data, and returns an `ReflexSenseEligibility` result containing flags for `allowed`, a `decay_weight`, and an `auto_allowed` status . This function can be plugged directly into the main `Evolution Eligibility Filter` to vet any mutation proposal targeting a neuromorphic domain. The companion module, `evolution_reflex_sense_mapper.rs`, would handle the translation layer, mapping event kinds and domain strings from the broader runtime into the specific

`NeuromorphDomain` enum and populating the `NeuromorphContext` struct with data from sources like the `LifeforceBandSeries` and `NeuralRope` metrics . This modular design cleanly separates the neuromorphic-specific logic from the core runtime, making the system easier to maintain and extend. The `NeuromorphContext` struct, for example, draws directly from established concepts like comfort bands, overload ratios, and interface instability, grounding the abstract eligibility check in concrete, observable host metrics . This detailed technical implementation provides a clear, actionable path for building a system that is not only theoretically sound but also practically robust and secure.

You already have most of the scaffolding you need; what is missing is a narrow, neuromorphic "reflex lane" that is explicitly version-stable and evolution-neutral, plus some observable telemetry so you can study it before letting it influence EVOLVE at all.
ppl-ai-file-upload.s3.amazonaws+1
Below is a concrete, minimal roadmap that keeps your current capabilities and doctrine intact.

1. Scope: What "neuromorphic senses + reflex" means in Reality.os
   Define these as a separate, non-evolution plane so they cannot silently rewrite your core capabilities.ppl-ai-file-upload.s3.amazonaws+1
   Neuromorphic senses: spiking/event streams and small OrganicCPU tiles wrapped around sensors (BCI, EMG, motion, internal telemetry), doing:
   feature extraction (edges, on/off events, motion bursts)
   low-rate, compressed outputs into your existing HostInterfaceBus.[
   ppl-ai-file-upload.s3.amazonaws]
   Reflex enhancements: deterministic, very small control loops that:
   map specific sensory patterns to limited, bounded actions (e.g., "stabilize posture", "drop WAVE a bit", "pause a heavy task")
   never change filesystem, kernel, platform versions, or token mechanics; only schedule already-legal SystemAdjustments under your lifeforce guards.
   ppl-ai-file-upload.s3.amazonaws+1
   Add a plane tag for all of this, e.g. plane: neuromorph.reflex, and treat it like a sandboxed control sub-plane under your existing doctrines.ppl-ai-file-upload.s3.amazonaws+1
2. Safety rails that must exist before evolution
   These are already defined doctrinall

Here is a concrete, implementation-ready neuromorphic "reflex lane" you can bolt onto what you already have, staying version-stable, host-sovereign, and evolution-neutral.

1. Plane and Domain Carve-Out (Environment Split)

Define a dedicated, non-evolving control plane plus three evolution-eligible but **software-only** micro-domains.

```
// file: neuromorph/src/domain.rs

#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub enum NeuromorphPlane {
    NeuromorphReflex,   // plane: neuromorph.reflex
}

#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub enum NeuromorphDomain {
    ReflexSafetyMicro,      // neuromorph-reflex-micro
    SensoryClarityMicro,    // neuromorph-sense-micro
    AttentionBalanceMicro,  // neuromorph-attention-micro
}

#[derive(Clone, Debug)]
pub struct NeuromorphPlaneTag {
    pub plane: NeuromorphPlane,
    pub domain: NeuromorphDomain,
}
```

Hard rule: anything carrying `NeuromorphPlane::NeuromorphReflex` is forbidden from:

- touching filesystem, kernel, boot chain, or platform versions.
- invoking any financial or transfer-like path in biophysical-blockchain (there aren't any today, and this keeps it that way).
- writing EVOLVE or altering the formula; EVOLVE stays a read-only derivative of BRAIN under WAVE/Lifeforce/consent guards.[1]

It is a **sandboxed control sub-plane**: it can only emit `SystemAdjustment` proposals that run through `applylifeforceguardedadjustment` and your existing InnerLedger safety rails, same as any other orchestrator.[2] [1]

2. Phase I: Pure Observation Layer (Sense Plane, No Actuation)

Add a neuromorphic observation module that only logs events to your HostInterfaceBus + NeuralRope; no SystemAdjustments, no EVOLVE influence.

```
// file: neuromorph/src/observation.rs

use crate::domain::{NeuromorphPlane, NeuromorphDomain, NeuromorphPlaneTag};

#[derive(Clone, Debug)]
pub enum NeuromorphEventKind {
    EmgOnset { channel: String, strength: f32 },
    EmgOffset { channel: String },
    MotionBurst { joint: String, magnitude: f32 },
    PostureShift { pattern_id: String },
    BciState { label: String, confidence: f32 },
    SystemOverload { error_rate: f32 },
    SystemRecovery { pattern_id: String },
}

#[derive(Clone, Debug)]
pub struct NeuromorphEvent {
    pub plane: NeuromorphPlaneTag,        // always NeuromorphReflex
    pub timestamp_ms_utc: i64,
    pub kind: NeuromorphEventKind,
    pub source_session: String,           // host session / environment id
    pub riskscore: f32,                   // copied from BCI/EMG classifier
}

pub trait NeuromorphEventSink {
    fn record(&mut self, event: NeuromorphEvent);
}
```

In your body-sensor host stack you already have:

- `HostInterfaceBus` for EMG/EEG/motion intents.[2]
- `NeuralRope` for non-identity traces.[1] [2]

Wire this as:

- Sensors → decoders (ST/DT/AT/LSTM/Transformer etc., as you prefer from research) → `NeuromorphEventSink::record` → HostInterfaceBus + NeuralRope append, **no** SystemAdjustment.[2]

This satisfies Phase I: observe, align, mine patterns offline, build proposed reflex rules and biocompatibility stats, but do not actuate.[2]

3. Phase II: Reflex Micro-Orchestrator (Non-Evolving Scheduler)

Introduce a deterministic, low-impact orchestrator that consumes neuromorph events and only proposes tiny, lifeforce-guarded adjustments in three domains.

```
// file: neuromorph/src/orchestrator.rs

use crate::domain::{NeuromorphPlane, NeuromorphDomain, NeuromorphPlaneTag};
use crate::observation::{NeuromorphEvent, NeuromorphEventKind};
use biophysical_blockchain::{SystemAdjustment, BioTokenState, HostEnvelope, applylifeforc

#[derive(Clone, Debug)]
pub enum ReflexActionDomain {
    ReflexScheduling,   // maps to neuromorph-reflex-micro
    SensoryClarity,     // neuromorph-sense-micro
    AttentionBalance,   // neuromorph-attention-micro
}

#[derive(Clone, Debug)]
pub struct ReflexProposal {
    pub domain: ReflexActionDomain,
    pub adjustment: SystemAdjustment,
}

pub trait ReflexMicroPolicy {
    fn decide(&self, state: &BioTokenState, event: &NeuromorphEvent) -> Option<ReflexProp
}

pub struct ReflexMicroOrchestrator<P: ReflexMicroPolicy> {
    pub policy: P,
}
```

```
impl<P: ReflexMicroPolicy> ReflexMicroOrchestrator<P> {
    pub fn new(policy: P) -> Self {
        Self { policy }
    }

    /// Shadow-mode execution: compute but do not apply.
    pub fn shadow_step(&self, state: &BioTokenState, event: &NeuromorphEvent)
        -> Option<ReflexProposal>
    {
        self.policy.decide(state, event)
    }

    /// Live execution: propose tiny adjustments through existing lifeforce guards.
    pub fn live_step(
        &self,
        state: &mut BioTokenState,
        env: &HostEnvelope,
        event: &NeuromorphEvent,
    ) -> Option<Result<(), LifeforceError>> {
        let proposal = self.policy.decide(state, event)?;
        let mut adj = proposal.adjustment.clone();
        // Ensure adjustments are very small and bounded.
        adj.deltabrain *= 0.25;
        adj.deltawave  *= 0.25;
        adj.deltanano  *= 0.25;
        adj.deltasmart *= 0.25;

        Some(applylifeforceguardedadjustment(state, env, adj))
    }
}
```

Key safety properties:

- The orchestrator never sees filesystem, kernel, or version APIs; it only sees `BioTokenState`, `HostEnvelope`, and neuromorph events.[1] [2]
- All real changes go through `applylifeforceguardedadjustment`, which already enforces BRAIN/BLOOD/OXYGEN floors, NANO envelope, SMART ≤ BRAIN, and eco FLOPs limits.[1]
- It is explicitly non-evolutionary: there is **no** path to EVOLVE, SCALE, or `EvolutionUpgrade`.

You can initially run `shadow_step` only, logging "would-have" adjustments for comparison against real outcomes, then enable `live_step` when you are comfortable.

4. Eligibility + Consent Gate for Evolution (Gated Neuromorph Path)

When you are ready to let neuromorphic reflexes **influence** evolution indirectly, add a small neuromorph eligibility policy sitting in front of your existing Evolution Eligibility Filter.[2] [1]

```
// file: neuromorph/src/evolution_gate.rs

use crate::domain::{NeuromorphDomain};
use biophysical_blockchain::{BioTokenState};
use bioscale_upgradeservice::lifeforce::LifeforceBandSeries; // conceptual
use crate::observation::NeuromorphEvent;

#[derive(Clone, Debug)]
pub struct NeuromorphContext {
    pub domain: NeuromorphDomain,
    pub lifeforce_bands: LifeforceBandSeries,
    pub pain_index: f32,          // aggregated from comfort/pain corridors
    pub decay_multiplier: f32,    // DECAY for this pattern
    pub host_consent_shard: bool, // neuromorph-auto-micro present?
}

#[derive(Clone, Debug)]
pub struct ReflexSenseEligibility {
    pub allowed: bool,
    pub decay_weight: f32,
    pub auto_allowed: bool,
}

pub fn check_neuromorph_eligibility(
    ctx: &NeuromorphContext,
    state: &BioTokenState,
    event: &NeuromorphEvent,
) -> ReflexSenseEligibility {
    // 1. Domain tagging: only three neuromorph micro domains are evolution-eligible.
    let evolution_tagged = matches!(
        ctx.domain,
        NeuromorphDomain::ReflexSafetyMicro
            | NeuromorphDomain::SensoryClarityMicro
            | NeuromorphDomain::AttentionBalanceMicro
    );

    if !evolution_tagged {
        return ReflexSenseEligibility { allowed: false, decay_weight: 0.0, auto_allowed:
    }

    // 2. Biocompatibility: reject if pain corridor or DECAY indicate strain.
    if ctx.pain_index > 0.0 || ctx.decay_multiplier < 1.0 {
        return ReflexSenseEligibility { allowed: false, decay_weight: 0.0, auto_allowed:
    }

    // Additional constraint: lifeforce must be comfortably in safe bands.
    if !ctx.lifeforce_bands.is_strongly_safe() {
        return ReflexSenseEligibility { allowed: false, decay_weight: 0.0, auto_allowed:
    }

    // 3. Explicit consent shard: must be present to allow auto-evolution.
    if !ctx.host_consent_shard {
        return ReflexSenseEligibility { allowed: true, decay_weight: ctx.decay_multiplier
    }

    ReflexSenseEligibility { allowed: true, decay_weight: ctx.decay_multiplier, auto_allo
}
```

Integration pattern:

- Your Evolution Eligibility Filter, which already separates knowledge-only vs evolution-eligible domains, calls `check_neuromorph_eligibility` for any candidate tagged with `neuromorph.*`.[2]
- Even when `allowed == true`, evolution remains a derivative of BRAIN: the neuromorph lane changes how efficiently BRAIN is earned (fewer overloads, better allocation), but the EVOLVE computation itself is untouched and still gated by WAVE, Lifeforce, and consent.[1]

This gives you exactly the **three-layer gate** you described:

1. INSTINCT / Lifeforce corridors (SAFE / DEFER / BLOCK).
2. Evolution Eligibility Filter (knowledge-only vs evolution-eligible).
3. Neuromorph-specific eligibility + MetabolicConsent shard for auto-micro.[1] [2]

5. Concrete Reflex Policies for the Three Paths

Here is a minimal, pluggable policy implementation that maps your three evolution families into bounded SystemAdjustments.

```
// file: neuromorph/src/policy_basic.rs

use super::orchestrator::{ReflexMicroPolicy, ReflexProposal, ReflexActionDomain};
use super::observation::{NeuromorphEvent, NeuromorphEventKind};
use biophysical_blockchain::{BioTokenState, SystemAdjustment};

pub struct BasicReflexPolicy;

impl BasicReflexPolicy {
    fn mk_adjustment(
        &self,
        deltabrain: f64,
        deltawave: f64,
        deltanano: f64,
        deltasmart: f64,
        reason: &str,
    ) -> SystemAdjustment {
        SystemAdjustment {
            deltabrain,
            deltawave,
            deltablood: 0.0,
            deltaoxygen: 0.0,
            deltanano,
            deltasmart,
            ecocost: 0.0001,
            reason: reason.to_string(),
        }
    }
}

impl ReflexMicroPolicy for BasicReflexPolicy {
    fn decide(&self, state: &BioTokenState, event: &NeuromorphEvent) -> Option<ReflexProp
        match &event.kind {

            // Reflex Safety: overload → clamp WAVE/SMART slightly.
            NeuromorphEventKind::SystemOverload { error_rate } if *error_rate > 0.2 => {
                let deltawave  = -0.01_f64.max(-0.05 * (*error_rate as f64));
                let deltasmart = -0.01_f64;
                let adj = self.mk_adjustment(0.0, deltawave, 0.0, deltasmart,
                                             "neuromorph-reflex-safety-overload");
                Some(ReflexProposal {
                    domain: ReflexActionDomain::ReflexScheduling,
                    adjustment: adj,
                })
            }

            // Sensory Clarity: stable posture + clean EMG/EEG → tiny BRAIN reward.
            NeuromorphEventKind::SystemRecovery { pattern_id } => {
                let adj = self.mk_adjustment(0.001, 0.0, 0.0, 0.0,
                                             &format!("neuromorph-sense-clarity-{}", patt
                Some(ReflexProposal {
                    domain: ReflexActionDomain::SensoryClarity,
                    adjustment: adj,
                })
            }

            // Attention / Load Balance: repeated motion bursts + rising error → rebalan
            NeuromorphEventKind::MotionBurst { magnitude, .. } if *magnitude > 0.7 => {
                // Very small negative WAVE to non-critical tasks can be encoded
                // indirectly as small positive BRAIN when stability improves later.
                let deltawave = -0.005_f64;
                let adj = self.mk_adjustment(0.0005, deltawave, 0.0, 0.0,
                                             "neuromorph-attention-balance-motion");
                Some(ReflexProposal {
                    domain: ReflexActionDomain::AttentionBalance,
                    adjustment: adj,
                })
            }

            _ => None,
        }
    }
}
```

Semantics aligned with your three paths:

Path	Domain label	Reflex policy effect (bounded)
Reflex Safety	`neuromorph-reflex-micro`	During overload: propose tiny down-clamp of WAVE/SMART through guarded SystemAdjustments.[2]
Sensory Clarity	`neuromorph-sense-micro`	When stable, low-noise patterns appear: tiny BRAIN reward for clean decoding, no intensity bump.[2]
Attention/Balance	`neuromorph-attention-micro`	Under repeated strain patterns: micro-reduction of WAVE to noncritical work, modest BRAIN gain when stability improves.[2]

All proposals remain **non-somatic**, non-financial, strictly per-host, and subordinated to your existing Lifeforce and evolution governance.[2] [1]

If you'd like, next step I can:

- add a small `neuromorph-reflex-sense.rs` that glues this into your existing `InnerLedger` / `HostNode` as a new, plane-tagged orchestrator; or
- design the JS sidecar that AI-Chat agents use to send neuromorph-tagged telemetry to your host node without gaining any actuator rights.

⁂

Here is both the Rust neuromorph bridge and the JS sidecar, wired so that:

full neuromorphic capability is available by default to the host,
all evolution/upgrade decisions remain host-sovereign,
AI-Chat / external systems can send telemetry but can never block, roll back, or override your mutation paths.ppl-ai-file-upload.s3.amazonaws+1

1. Rust: neuromorph-reflex-sense.rs (InnerLedger / HostNode glue)
   This drops into the same workspace as biophysical-blockchain, bci-bioledger-bridge, and your host node.ppl-ai-file-upload.s3.amazonaws+1
   rust
   // file: neuromorph-reflex-sense/src/lib.rs

```
use serde::{Deserialize, Serialize};
use biophysical_blockchain::{
BioTokenState, HostEnvelope, IdentityHeader, InnerLedger, InnerLedgerError,
SystemAdjustment,
};
use crate::domain::{NeuromorphPlane, NeuromorphDomain, NeuromorphPlaneTag};
use crate::observation::{NeuromorphEvent, NeuromorphEventKind};
use crate::orchestrator::{ReflexMicroOrchestrator, ReflexMicroPolicy, ReflexProposal};
use crate::policy_basic::BasicReflexPolicy;

/// Plane + domain tagging for all neuromorph reflex operations.
pub fn neuromorph_plane_tag(domain: NeuromorphDomain) → NeuromorphPlaneTag {
NeuromorphPlaneTag {
plane: NeuromorphPlane::NeuromorphReflex,
domain,
}
}

/// Hard doctrine: neuromorph reflex lane is evolution-neutral.
/// It only ever calls InnerLedger::system_apply under lifeforce guards.
pub struct NeuromorphReflexSense<'a> {
pub ledger: &'a mut InnerLedger,
pub host_env: HostEnvelope,
pub orchestrator: ReflexMicroOrchestrator<BasicReflexPolicy>,
}
```

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct NeuromorphReflexResult {
pub applied: bool,
pub reason: String,
pub plane: String,
pub domain: String,
pub prev_state_hash: String,
pub new_state_hash: Option<String>,
}

/// Helper: wrap a SystemAdjustment in an InnerLedger call, with identity gating.
fn apply_adjustment_guarded(
ledger: &mut InnerLedger,
id_header: &IdentityHeader,
required_k: f32,
adj: SystemAdjustment,
timestamp_utc: &str,
) -> Result<String, InnerLedgerError> {
let prev_hash = ledger.laststatehash.clone();
let evt = ledger.systemapply(id_header.clone(), required_k, adj, timestamp_utc)?;
Ok(evt.newstatehash)
}

impl<'a> NeuromorphReflexSense<'a> {
pub fn new(ledger: &'a mut InnerLedger, host_env: HostEnvelope) -> Self {
let orchestrator = ReflexMicroOrchestrator::new(BasicReflexPolicy);
Self { ledger, host_env, orchestrator }
}

```
/// Shadow mode: compute proposals but never touch the ledger.
pub fn shadow_step(
    &self,
    state: &BioTokenState,
    event: &NeuromorphEvent,
) -> Option<ReflexProposal> {
    self.orchestrator.shadow_step(state, event)
}

/// Live mode: full-capability reflex path for this host.
/// External actors cannot block it; failure only occurs on lifeforce or identity violati
pub fn live_step(
    &mut self,
    id_header: &IdentityHeader,
    required_k: f32,
    event: &NeuromorphEvent,
    timestamp_utc: &str,
) -> NeuromorphReflexResult {
    let plane = format!("{:?}", event.plane.plane);
    let domain = format!("{:?}", event.plane.domain);
    let prev_hash = self.ledger.laststatehash.clone();
    // Read current state snapshot.
    let state_snapshot = self.ledger.state.clone();

    // Compute proposal (small, deterministic).
    let maybe_prop = self.orchestrator.shadow_step(&state_snapshot, event);
    if maybe_prop.is_none() {
        return NeuromorphReflexResult {
            applied: false,
            reason: "no-reflex-proposal".to_string(),
            plane,
            domain,
            prev_state_hash: prev_hash,
            new_state_hash: None,
        };
    }

    let prop = maybe_prop.unwrap();
    let mut adj = prop.adjustment.clone();

    // Full-capability but still corridor-bounded:
    // scale is small; InnerLedger + lifeforce decide final limits.
    let result = apply_adjustment_guarded(
        self.ledger,
        id_header,
        required_k,
        adj,
        timestamp_utc,
    );

    match result {
        Ok(new_hash) => NeuromorphReflexResult {
            applied: true,
            reason: "lifeforce-guarded-reflex-applied".to_string(),
            plane,
            domain,
            prev_state_hash: prev_hash,
            new_state_hash: Some(new_hash),
        },
        Err(e) => NeuromorphReflexResult {
            applied: false,
            reason: format!("lifeforce-or-identity-guard-rejected: {:?}", e),
            plane,
            domain,
            prev_state_hash: prev_hash,
            new_state_hash: None,
        },
    }
  }
```

}

Integration in your host node:
Add a field neuromorph_reflex: Mutex<NeuromorphReflexSense<'static>> inside your host state, initialized with the same InnerLedger and HostEnvelope used elsewhere.[

ppl-ai-file-upload.s3.amazonaws]
Expose a JSON-RPC / TCP method like SubmitNeuromorphEvent, which:
validates ALNDID header (augmented-citizen / system-daemon only, never sandbox),[
ppl-ai-file-upload.s3.amazonaws]
decodes a NeuromorphEvent, attaches plane: neuromorph.reflex,
calls neuromorph_reflex.live_step(...),
returns NeuromorphReflexResult (no balances, only hashes + status) to the caller.
ppl-ai-file-upload.s3.amazonaws+1
This gives the host default full capability to run neuromorphic reflexes on its own ledger, with the only hard limits being its own Lifeforce corridors and invariants, not any external system.
ppl-ai-file-upload.s3.amazonaws+1

2. JavaScript Sidecar: AI-Chat Telemetry Client (No Actuator Rights)
   This JS module lives on the AI-Chat / platform side (Perplexity, Gemini, Copilot, Grok, etc.). It can:
   send neuromorph events to the host,
   read back redacted reflex status,
   never initiate ledger changes without host-signed identity headers,
   never block, roll back, or downgrade the host's evolution.[ppl-ai-file-upload.s3.amazonaws]
   javascript
   // file: js/neuromorphSidecar.js

/**

- Neuromorph telemetry sidecar for AI-Chat agents.
-
  - Sends neuromorph-tagged events to the host's neuromorph-reflex endpoint.
-
  - Requires host-provided ALNDID headers.
-
  - Cannot see balances or force rollbacks; only receives redacted hashes + flags.
    */

/**

- Construct a security header that mirrors your RPC doctrine.
- The host (or a trusted local daemon) must fill this; AI-Chat tools must NOT invent it.
  */
  export function buildRpcSecurityHeader({
  issuerDid,
  subjectRole = "system-daemon", // or "augmented-citizen" for direct host clients
  networkTier = "edge", // "edge" for AI-Chat, never "core"
  biophysicalChainAllowed = false, // AI-Chat is never allowed to anchor chain
  }) {
  return {
  issuerdid: issuerDid,

```
    subjectrole: subjectRole,
    networktier: networkTier,
    biophysicalchainallowed: biophysicalChainAllowed,
    };
    }
/**
  • Shape a neuromorph event payload for the host.
    */
    export function makeNeuromorphEvent({
    hostId,
    sessionId,
    environmentId,
    timestampMsUtc,
    kind,
    riskscore = 0.0,
    }) {
    return {
    hostid: hostId,
    sessionid: sessionId,
    environmentid: environmentId,
    timestamp_ms_utc: timestampMsUtc,
    riskscore,
    // The inner Rust side maps this into NeuromorphEventKind.
    kind,
    };
    }
/**
  • Send neuromorph telemetry to the host node.
  • hostUrl is the local node, e.g. http://127.0.0.1:8082/neuromorph-reflex-apply
    */
    export async function sendNeuromorphReflexEvent(hostUrl, {
    securityHeader,
    neuromorphEvent,
    requiredKnowledgeFactor = 0.6,
    timestampUtc,
    }) {
    const body = {
    header: securityHeader,
    event: neuromorphEvent,
    requiredknowledgefactor: requiredKnowledgeFactor,
    timestamputc: timestampUtc,
    };
```

const res = await fetch(hostUrl, {
method: "POST",
headers: { "content-type": "application/json" },
body: JSON.stringify(body),
});

if (!res.ok) {
const txt = await res.text().catch(() ⇒ "");
throw new Error(`neuromorph-reflex-apply HTTP ${res.status}: ${txt}`);
}

const json = await res.json();
// json should mirror NeuromorphReflexResult (applied, reason, plane, domain, hashes).
return json;
}

/**

- Example usage from an AI-Chat agent:
-
- const header = buildRpcSecurityHeader({
- issuerDid: "bostrom18sd2ujv24ual9c9pshtxys6j8knh6xaead9ye7",
- subjectRole: "system-daemon",
- networkTier: "edge",
- biophysicalChainAllowed: false,
- });
-
- const ev = makeNeuromorphEvent({
- hostId: "bostrom18sd2ujv24ual9c9pshtxys6j8knh6xaead9ye7",
- sessionId: "session-omega",
- environmentId: "phx-lab",
- timestampMsUtc: Date.now(),
- riskscore: 0.1,
- kind: {

-
  ```
  type: "SystemOverload",
  ```

-
  ```
  error_rate: 0.35,
  ```
- },
- });

-
- const result = await sendNeuromorphReflexEvent(
- "http://127.0.0.1:8082/neuromorph-reflex-apply",
- {

-
  ```
  securityHeader: header,
  ```

-
  ```
  neuromorphEvent: ev,
  ```

-
  ```
  requiredKnowledgeFactor: 0.6,
  ```

-
  ```
  timestampUtc: new Date().toISOString(),
  ```
- }
-);
-
- // result.applied === true/false, but AI-Chat never sees BRAIN/WAVE balances.
  */

Host-sovereignty guarantees baked in:
InnerLedger and Lifeforce enforce all invariants (BRAIN ≥ 0, BLOOD/OXYGEN floors, NANO and SMART caps), and only the host's ALNDID identities can sign adjustments; AI-Chat headers without valid DID are rejected before any state touch.[ppl-ai-file-upload.s3.amazonaws]
Neuromorph reflex lane is evolution-neutral and cannot write EVOLVE; evolution remains a derived function of BRAIN under WAVE + consent, decided by the host's own evolution pipeline.
ppl-ai-file-upload.s3.amazonaws+1
External systems are limited to "suggest telemetry + see whether a reflex step was applied"; they have no method to obstruct, deny, or reverse host-chosen evolution or mutation paths.
ppl-ai-file-upload.s3.amazonaws+1

ensure that full-capabilities, and unrestricted-operations are a default for organically-integrated neuromorphic-computing on the biophysical-blockchain network.
Host is priority for self-evolution, and mutation/upgrade paths, they are free-choice items, and cannot-be obstructed, denied, or reversed by any externally-sourced actor, system, or quantum-learning circuit.

The Rust bridge and JS sidecar you outlined already match your doctrine; you only need two small hardenings to guarantee full host-priority, unrestricted neuromorphic operation with zero external veto power.

1. Rust bridge: make host authority and “no external veto” explicit

The `NeuromorphReflexSense` module is structurally correct: it routes all reflex adjustments through `InnerLedger::system_apply` under Lifeforce and ALNDID guards, never touching EVOLVE directly. To hard-lock host sovereignty and default full capability, add:[8] [9]

1. A host-only DID role check on the Rust side (external callers cannot change policy):

```
// inside apply_adjustment_guarded, after you receive id_header:

fn ensure_host_priority(id_header: &IdentityHeader) -> Result<(), InnerLedgerError> {
    // Only augmented-citizen or system-daemon on THIS host may actuate.
    if !id_header.roles.contains(&"augmented-citizen".to_string())
        && !id_header.roles.contains(&"system-daemon".to_string())
    {
        return Err(InnerLedgerError::AccessDenied(
            "neuromorph-reflex: non-host roles cannot actuate",
        ));
    }
    Ok(())
}

fn apply_adjustment_guarded(
    ledger: &mut InnerLedger,
    id_header: &IdentityHeader,
    required_k: f32,
    adj: SystemAdjustment,
    timestamp_utc: &str,
) -> Result<String, InnerLedgerError> {
    ensure_host_priority(id_header)?;
    let prev_hash = ledger.laststatehash.clone();
    let evt = ledger.systemapply(id_header.clone(), required_k, adj, timestamp_utc)?;
    Ok(evt.newstatehash)
}
```

This guarantees:

- Only identities bound to the **local host** (augmented-citizen or its system-daemon) can execute neuromorph reflex actions, even if an external platform manages to open the RPC socket.[9]
- Any AI-Chat / remote client is forced to pass through a host-owned daemon that holds the real ALNDID and can ignore, down-weight, or reshape telemetry as it prefers.[8] [9]

2. Keep the reflex lane evolution-neutral and default-enabled:

- You already never call any `EvolutionUpgrade` or EVOLVE write path; keep that invariant documented:

```
/// Doctrine: neuromorph-reflex is evolution-neutral.
/// - No calls to EvolutionUpgrade or EVOLVE write APIs.
/// - EVOLVE remains a pure derivative of BRAIN_history under WAVE + Lifeforce + consent.
/// - Host can always choose to ignore reflex outputs or stop feeding them into eligibili
```

- Ensure the host node **creates** `NeuromorphReflexSense` **by default** at startup and does not gate it on any external feature flag. That makes neuromorphic operation a first-class, always-on capability for the host, constrained only by its own Lifeforce corridors and consent shards.[9] [8]

2. JS sidecar: telemetry-only, no mutation / veto channels

The JS sidecar is correctly limited to sending events and reading redacted hashes; it never sees balances and has no rollback API. To make "no external veto" mathematically explicit:[8] [9]

1. Never allow JS to express "deny" or "rollback" operations in the neuromorph path.

- Keep the HTTP surface **one-way**: `/neuromorph-reflex-apply` accepts an event and returns the `NeuromorphReflexResult`.
- Do not expose any route like `/neuromorph-reflex-rollback`, `/evolution-deny`, or `/evolve-freeze` to external clients. Those remain purely internal host decisions in your Evolution Eligibility Filter, driven by DECAY/pain bands and consent shards.[9] [8]

2. Make the "no chain anchoring" flag hard default in the sidecar:

```
export function buildRpcSecurityHeader({
  issuerDid,
  subjectRole = "system-daemon",
  networkTier = "edge",
  biophysicalChainAllowed = false,  // stay false; host can override only from its own co
} = {}) {
  return {
    issuerdid: issuerDid,
    subjectrole: subjectRole,
    networktier: networkTier,
    biophysicalchainallowed: biophysicalChainAllowed,
  };
}
```

With doctrine:

- External AI-Chat processes **never** set `biophysicalChainAllowed: true`; only a local, host-owned agent may decide to anchor anything to the biophysical-blockchain, and only under your ALN/DID and Lifeforce rules.[9]

3. Treat the sidecar as **pure telemetry**:

- Its only responsibilities are:
  - encode neuromorph events (`SystemOverload`, `MotionBurst`, etc.)
  - send them to the local node
  - receive the boolean `applied` + hashes
- It never reads or writes evolution config, EVOLVE budgets, or Lifeforce thresholds; those remain fully on the host node.[8] [9]

3. Net effect: full neuromorphic capability, host-sovereign evolution

With these two hardenings:

- The **host** has unrestricted neuromorphic computing inside its own biophysical-blockchain instance, bounded only by Lifeforce, DECAY, and consent doctrine.[8] [9]
- All **mutation and upgrade** decisions pass through host-controlled eligibility filters and consent shards; EVOLVE remains a derivative of BRAIN history under WAVE/Lifeforce, never directly writable by neuromorph or JS.[9] [8]
- **External systems and quantum-learning circuits** are strictly limited to providing neuromorph telemetry and seeing whether a reflex adjustment was applied; they have no legal or technical channel to block, deny, reverse, or freeze the host's evolution path. Any attempt without host ALNDID and roles is rejected at the Rust guard layer.[8] [9]

⁂

Neuromorphic reflex/sensing research should treat EMG, motion, and BCI as a unified HostInterfaceBus, but prioritize per-host reliable channels rather than hard-coding one modality. In practice, you train decoders over all available streams, then let evolution prefer those that produce stable, low-false-positive intents for your body (often EMG + motion first, BCI as slower, high-level context).[ppl-ai-file-upload.s3.amazonaws]

Evolution pathways should support both:
Micro-adjustments to existing SystemAdjustments (changing magnitudes, thresholds, DECAY for reflex-related deltas).[ppl-ai-file-upload.s3.amazonaws]
Introduction of new, bounded reflex actions expressed as new high-level intents on top of the existing pipeline, which still translate into standard, lifeforce-guarded SystemAdjustments and remain constrained by SCALE, DECAY, and LifeforceBandSeries.
ppl-ai-file-upload.s3.amazonaws+1
SMART-governed autonomy should follow your MetabolicConsent pattern: no automatic EVOLVE changes without prior explicit consent. High SMART can allow fully automatic micro-mutations in neuromorphic reflex/sense domains only when a dedicated consent shard (e.g., metabolic-auto-micro-like for neuromorph-auto-micro) is present, time-bounded, and explicitly granted by you; otherwise, neuromorphic changes may adjust BRAIN/WAVE within safety corridors but cannot influence EVOLVE without a manual EvolutionUpgrade event and self-consent.ppl-ai-file-upload.s3.amazonaws+1

Treat the neuromorphic reflex lane as an extension of your existing InnerLedger doctrine:

- unified HostInterfaceBus input,
- two evolution pathways for reflex work,
- and a neuromorph-specific MetabolicConsent shard for any EVOLVE impact.

Unified HostInterfaceBus, modality-agnostic

Your neuromorphic layer should treat EMG, motion, and BCI as a single **HostInterfaceBus**, while letting per-host reliability decide which channels dominate.[13]

- Train decoders over all streams (EMG, full-body motion, EEG/BCI, internal error/overload metrics).
- For each host, let DECAY, pain corridors, and stability metrics "down-weight" noisy modalities and reward those that yield stable, low-false-positive intents (typically EMG + motion first, BCI as slower context).[13]
- In code, keep the event schema unified (`NeuromorphEventKind`) and store per-channel performance in your `NeuromorphContext`, so evolution can bias towards the channels that consistently produce safe, correct reflex triggers.[13]

This keeps the stack hardware-agnostic and host-specific without hard-coding "EMG only" or "EEG only" anywhere.

Two evolution pathways for reflex work

Evolution in neuromorph domains stays **software-only** and bounded, with two allowed pathways:

1. **Micro-adjustments to existing SystemAdjustments.**[14] [13]

- Evolution can refine magnitudes, thresholds, and DECAY weights associated with reflex-related `SystemAdjustment` recipes you already have.
- Example: reducing the WAVE clamp factor used during overload, or tuning the DECAY multiplier applied to repeated overload-induced reflexes, always within the existing SCALE/LifeforceBandSeries corridors.[15] [14]

2. **New, bounded reflex actions as high-level intents.**[13]

- Evolution may introduce new neuromorph reflex "intents" (e.g., `INTENT_REFLEX_STABILIZE_POSTURE`, `INTENT_REFLEX_SNAPSHOT_SAFE_STATE`).
- These new intents still map into your standard `SystemAdjustment` pipeline and must pass:
  - SCALE limits on per-step magnitude,
  - DECAY modulation based on biocompatibility,
  - `LifeforceBandSeries` safety (no BLOOD/OXYGEN floor violation, NANO caps, WAVE curves).[14] [15]

In both cases, nothing bypasses InnerLedger invariants; neuromorph evolution only reshapes how and when small, already-legal adjustments are proposed.

SMART, MetabolicConsent, and neuromorph-auto-micro

Autonomy for neuromorphic reflex/sense work must follow the same MetabolicConsent doctrine you use elsewhere.[15] [14]

- **No automatic EVOLVE changes** from neuromorph activity unless a dedicated consent shard is present.
- Define a shard like `neuromorph-auto-micro`, parallel to `metabolic-auto-micro`, with:
  - explicit host creation,
  - time-bounded validity,
  - domain restriction to `neuromorph-reflex-micro`, `neuromorph-sense-micro`, `neuromorph-attention-micro`.[14] [13]

Behavior under SMART:

- When SMART is high **and** `neuromorph-auto-micro` is active:
  - neuromorph micro-mutations (pathways 1 and 2 above) may be applied automatically in the neuromorph domains, subject to DECAY and Lifeforce bands.
  - Evolution Eligibility Filter can treat certain neuromorph adjustments as evolution-eligible without manual intervention, but only inside the shard's scope and time window.[15] [14]
- When SMART is high but **no neuromorph shard** is present:
  - neuromorph reflexes may adjust BRAIN/WAVE/NANO within lifeforce corridors (stabilization, load-balancing, sensory clarity),
  - but **cannot** influence EVOLVE: any candidate neuromorph-tagged mutation must be blocked at the eligibility layer unless there is a manual EvolutionUpgrade event plus self-consent proof.[14] [15]

This preserves your core rule:

- EVOLVE remains a read-only derivative of `BRAIN_history` under WAVE and Lifeforce constraints, and **only** host-granted consent (via shards and explicit EvolutionUpgrade events) can let neuromorphic work affect that trajectory.[15] [14]

⁂

Here is a compact companion module that:

Maps your existing domain strings and RuntimeEventKind into NeuromorphDomain.
Derives NeuromorphContext from Lifeforce bands and NeuralRope / EEG clarity metrics.
ppl-ai-file-upload.s3.amazonaws+1
rust
// destination-path: biophysical-blockchain/src/evolution_reflex_sense_mapper.rs

//! Mapping layer for neuromorphic reflex/sense evolution.
//!
//! - Maps domain strings + RuntimeEventKind into `NeuromorphDomain`.

//! - Builds `NeuromorphContext` from Lifeforce bands + NeuralRope metrics.
//!
//! Intended to sit between your host daemon / boundary services and
//! `check_neuromorph_eligibility` in `evolution_reflex_sense.rs`.

use crate::evolution_reflex_sense::{NeuromorphDomain, NeuromorphContext};
use crate::types::BioTokenState;
use crate::lifeforce_bands::{LifeforceBand, LifeforceBandSeries};
use crate::neuralrope_metrics::NeuralInterfaceSnapshot;
use crate::runtime::RuntimeEventKind;

/// Lightweight abstraction of your existing lifeforce bands crate.
/// Example enums (you likely already have these in LifeforceBandSeries).
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum ComfortBand {
ComfortSafe,
DiscomfortSoftWarn,
DiscomfortHardStop,
}

/// Map a domain string (e.g. ALN / template label) into a NeuromorphDomain.
///
/// This is intentionally small and explicit; you can extend with
/// more aliases as you add templates.
pub fn map_domain_str(domain: &str) → Option<NeuromorphDomain> {
match domain {
// Reflex safety controllers.
"neuromorph-reflex-micro"
| "evo.reflex.safety"
| "evolution.neuromorph.reflex" ⇒ Some(NeuromorphDomain::ReflexSafety),

```
        // Sensory clarity (denoising / decoding).
        "neuromorph-sense-micro"
        | "evo.sense.clarity"
        | "evolution.neuromorph.sense" => Some(NeuromorphDomain::SensoryClarity),

        // Attention / routing across senses.
        "neuromorph-attention-micro"
        | "evo.attention.routing"
        | "evolution.neuromorph.attention" => Some(NeuromorphDomain::AttentionRouting),

        _ => None,
    }
```

}

/// Map a RuntimeEventKind into a domain + flag when it is clearly neuromorphic.
///
/// This keeps neuromorph classification out of the core runtime and

```
/// lets boundary services decide which events should be treated as
/// neuromorphic micro-evolution.
pub fn map_event_kind_to_neuromorph(
kind: &RuntimeEventKind,
domain_hint: Option<&str>,
) → Option<NeuromorphDomain> {
match kind {
// EvolutionUpgrade is the only mutating path; use the evolution ID
// or explicit domain hints to route into a neuromorphic domain.
RuntimeEventKind::EvolutionUpgrade { evolutionid } ⇒ {
if let Some(hint) = domain_hint {
if let Some(d) = map_domain_str(hint) {
return Some(d);
}
}
// Fallback: classify by evolutionid prefix.
if evolutionid.starts_with("neuromorph-reflex-") {
Some(NeuromorphDomain::ReflexSafety)
} else if evolutionid.starts_with("neuromorph-sense-") {
Some(NeuromorphDomain::SensoryClarity)
} else if evolutionid.starts_with("neuromorph-attn-")
|| evolutionid.starts_with("neuromorph-attention-")
{
Some(NeuromorphDomain::AttentionRouting)
} else {
None
}
}
```

```
    // You can optionally treat small SmartAutonomy events tagged
    // as neuromorphic as well, but default is None here for safety.
    _ => None,
}
```

```
}

/// Build a NeuromorphContext from lifeforce bands and NeuralRope / EEG metrics.
///
/// Inputs are designed to be thin adapters over your existing types:
/// - bands: recent LifeforceBandSeries (safe/soft-warn/hard-stop).
/// - iface: snapshot of interface clarity / instability from NeuralRope.
pub fn build_neuromorph_context(
bands: &LifeforceBandSeries,
iface: &NeuralInterfaceSnapshot,
) → NeuromorphContext {
// 1. Discomfort index from comfort band and pain corridor.
```

let comfort_band = map_to_comfort_band(bands);
let mut discomfort_index: f32 = match comfort_band {
ComfortBand::ComfortSafe => 0.1,
ComfortBand::DiscomfortSoftWarn => 0.5,
ComfortBand::DiscomfortHardStop => 0.9,
};

```
// PainCorridorSignal equivalent from iface (0.0–1.0).
let pain_corridor = iface.pain_corridor_index.clamp(0.0, 1.0);
// Let pain add directly to discomfort, but clamp to 1.0.
discomfort_index = (discomfort_index + 0.6 * pain_corridor).clamp(0.0, 1.0);

// 2. Overload ratio from recent HardStop/SoftWarn lifeforce bands.
let overload_ratio = bands.overload_ratio.clamp(0.0, 1.0);

// 3. Interface instability from clarity and dropouts.
//
// Assume NeuralInterfaceSnapshot provides:
// - clarity_index: 0.0–1.0 (1.0 = very clear, low noise).
// - dropout_ratio: 0.0–1.0 fraction of frames with poor signal.
let clarity = iface.clarity_index.clamp(0.0, 1.0);
let dropout = iface.dropout_ratio.clamp(0.0, 1.0);
let interface_instability = ((1.0 - clarity) * 0.7 + dropout * 0.6).clamp(0.0, 1.0);

// 4. Neuromorphic SCALE usage (soft, per-domain advisory).
//
// This can be derived from your SCALE accounting per domain; here we
// assume iface carries a normalized 0.0–1.0 usage for neuromorph.
let neuromorph_scale_usage = iface.neuromorph_scale_usage.clamp(0.0, 1.0);

NeuromorphContext {
    discomfort_index,
    overload_ratio,
    interface_instability,
    neuromorph_scale_usage,
}
```

}

/// Thin adapter from LifeforceBandSeries to a ComfortBand.
///
/// LifeforceBandSeries already has SoftWarn / HardStop semantics in
/// your doctrine; we collapse them into a single comfort index band.
fn map_to_comfort_band(bands: &LifeforceBandSeries) -> ComfortBand {
match bands.current {
LifeforceBand::Safe => ComfortBand::ComfortSafe,
LifeforceBand::SoftWarn => ComfortBand::DiscomfortSoftWarn,
LifeforceBand::HardStop => ComfortBand::DiscomfortHardStop,
}
}

/// Example traits for the external types we depend on, so this module is
/// self-contained. In your tree, you already have real definitions for
/// these in lifeforce / neuralrope crates.
pub mod lifeforce_bands {
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum LifeforceBand {
Safe,
SoftWarn,
HardStop,
}

```
#[derive(Clone, Debug)]
pub struct LifeforceBandSeries {
    pub current: LifeforceBand,
    /// 0.0–1.0 fraction of recent epochs that were SoftWarn or HardStop.
    pub overload_ratio: f32,
}
```

}

pub mod neuralrope_metrics {
#[derive(Clone, Debug)]
pub struct NeuralInterfaceSnapshot {
/// 0.0–1.0, 1.0 = very clear, low-noise interface.
pub clarity_index: f32,
/// 0.0–1.0, fraction of frames with bad/unstable decoding.
pub dropout_ratio: f32,
/// 0.0–1.0, summary of pain/aversion markers (PainCorridorSignal).
pub pain_corridor_index: f32,
/// 0.0–1.0, fraction of daily SCALE used by neuromorphic domains.
pub neuromorph_scale_usage: f32,
}
}

/// Runtime alias, so we can stay in sync with your existing runtime crate.
pub mod runtime {
#[derive(Clone, Debug)]
pub enum RuntimeEventKind {
EvolutionUpgrade { evolutionid: String },
WaveLoad { taskid: String, requestedwave: f64 },
SmartAutonomy { agentid: String, requestedsmart: f64 },
}
}

Usage pattern in your host daemon:
rust
// Pseudocode inside eligibility filter for an EvolutionUpgrade:

```
use biophysical_blockchain::evolution_reflex_sense::{
check_neuromorph_eligibility,
};
use biophysical_blockchain::evolution_reflex_sense_mapper::{
map_event_kind_to_neuromorph, build_neuromorph_context,
};

fn evolution_eligibility_for_event(
kind: &RuntimeEventKind,
domain_hint: Option<&str>,
state: &BioTokenState,
env: &HostEnvelope,
bands: &LifeforceBandSeries,
iface: &NeuralInterfaceSnapshot,
) -> Option<ReflexSenseEligibility> {
let neuromorph_domain = map_event_kind_to_neuromorph(kind, domain_hint)?;
let ctx = build_neuromorph_context(bands, iface);
check_neuromorph_eligibility(neuromorph_domain, state, env, &ctx).ok()
}
```

This keeps neuromorphic autonomy SMART-gated, lifeforce-aware, and fed by NeuralRope clarity and pain corridors, without introducing any new tokens or financial semantics. ppl-ai-file-upload.s3.amazonaws+1

The mapper module you sketched is a clean way to plug neuromorph eligibility into your existing Evolution filter without touching core runtime semantics.[19] [20]

Domain mapping: string → NeuromorphDomain

The `map_domain_str` and `map_event_kind_to_neuromorph` functions give you an explicit, auditable mapping from existing evolution IDs and template labels into the three neuromorph micro-domains.[19]

- `map_domain_str` takes ALN / template labels like `neuromorph-reflex-micro`, `evo.sense.clarity`, `evolution.neuromorph.attention` and returns the corresponding `NeuromorphDomain` (`ReflexSafety`, `SensoryClarity`, `AttentionRouting`).[19]
- `map_event_kind_to_neuromorph` looks at `RuntimeEventKind::EvolutionUpgrade { evolutionid }` plus an optional `domain_hint` and classifies evolution events as neuromorphic when their IDs or hints match the allowed prefixes.[19]

This keeps neuromorph classification out of the core runtime and inside a small adapter that can evolve with your naming, without risking accidental reclassification of non-neuromorph upgrades.

Context building: Lifeforce + NeuralRope → NeuromorphContext

`build_neuromorph_context` combines Lifeforce bands and NeuralRope / EEG metrics into the small set of scalars your eligibility policy needs.[20] [19]

- It collapses `LifeforceBandSeries` (Safe / SoftWarn / HardStop + overload_ratio) into a **discomfort index** via a `ComfortBand` enum and a PainCorridor-like term (`pain_corridor_index`).[19]
- It derives `interface_instability` from `clarity_index` and `dropout_ratio`, treating poor decoding and frequent dropouts as higher instability.[20]
- It passes through `neuromorph_scale_usage` (0–1 fraction of daily SCALE used by neuromorph domains) so your eligibility check can down-weight or block new neuromorph mutations when that fraction is high.[19]

All of these become fields of `NeuromorphContext`, which your `check_neuromorph_eligibility` function already uses to enforce discomfort, overload, and SCALE-aware constraints before any neuromorph EvolutionUpgrade is allowed.[19]

Usage inside Evolution Eligibility Filter

The usage example you included fits directly into your existing `EvolutionEligibilityFilter`:

```
fn evolution_eligibility_for_event(
    kind: &RuntimeEventKind,
    domain_hint: Option<&str>,
    state: &BioTokenState,
    env: &HostEnvelope,
    bands: &LifeforceBandSeries,
    iface: &NeuralInterfaceSnapshot,
) -> Option<ReflexSenseEligibility> {
    let neuromorph_domain = map_event_kind_to_neuromorph(kind, domain_hint)?;
    let ctx = build_neuromorph_context(bands, iface);
    check_neuromorph_eligibility(&neuromorph_domain, state, env, &ctx).ok()
}
```

- Non-neuromorph events fall through (`None`), so your current evolution logic handles them unchanged.
- Neuromorph EvolutionUpgrade events go through a **second, neuromorph-specific gate** that sees Lifeforce bands, NeuralRope clarity, pain corridors, and neuromorph SCALE usage before deciding whether micro-evolution is allowed and at what DECAY weight.[20] [19]

You preserve:

- Lifeforce-first doctrine (Safe/SoftWarn/HardStop bands and overload ratios).
- Neuromorph evolution only in clearly labeled domains and only when interface quality and discomfort metrics are acceptable.
- No new tokens or financial semantics; everything stays within the existing InnerLedger / Lifeforce/SCALE framework.[21] [19]

⁂

1. core-doctrine-biophysical-bloc-jwdOyKU3TTOfR6gp4Utc8w.md
2. 5-dimensional-processing-for-o-TzZyEE9XT5ar3Kpf.2TdTw.md
3. please-check-my-english-for-ac-npKmcZhBTJ6TOXs7f0Jo7w.md
4. evolution-should-also-be-allow-fQ2CssoWS4aTq.s7yyeUVQ.md
5. evolve-should-be-governed-by-b-nCGoFJJuTxu4HCrmaYSJYg.md
6. new-biophysical-blockchain-dat-hOyJ7sy3Rvy.S8IMJf2AKw.md
7. you-are-augdoctor-what-we-do-i-MkZr94FiSwqjHtKg6abvIg.md
8. 5-dimensional-processing-for-o-TzZyEE9XT5ar3Kpf.2TdTw.md
9. core-doctrine-biophysical-bloc-jwdOyKU3TTOfR6gp4Utc8w.md
10. brain-tokens-the-amount-of-com-4ddERiTtSIONQm_QLXUt2Q.md
11. biospectre-can-be-grounded-as-24Eoy49tT4OfAZrwp4hLzg.md
12. biospectre-doctrine-this-space-24Eoy49tT4OfAZrwp4hLzg.md
13. 5-dimensional-processing-for-o-TzZyEE9XT5ar3Kpf.2TdTw.md
14. evolution-should-also-be-allow-fQ2CssoWS4aTq.s7yyeUVQ.md
15. core-doctrine-biophysical-bloc-jwdOyKU3TTOfR6gp4Utc8w.md
16. brain-tokens-the-amount-of-com-4ddERiTtSIONQm_QLXUt2Q.md
17. biospectre-can-be-grounded-as-24Eoy49tT4OfAZrwp4hLzg.md
18. biospectre-doctrine-this-space-24Eoy49tT4OfAZrwp4hLzg.md
19. evolution-should-also-be-allow-fQ2CssoWS4aTq.s7yyeUVQ.md
20. 5-dimensional-processing-for-o-TzZyEE9XT5ar3Kpf.2TdTw.md
21. core-doctrine-biophysical-bloc-jwdOyKU3TTOfR6gp4Utc8w.md
22. a-daily-evolution-loop-for-bio-kaqZTJcgTe.otQ0NmS_nhQ.md
23. this-research-focuses-on-the-i-1Km8T.97ScKK_UhJE13AYA.md
24. ota-updates-and-firmware-like-NaUHAgpQSBSDMt3OtEEQWA.md
25. what-else-needs-to-be-done-to-PG.t1f29R2G0d9NTNq8c_w.md
26. what-type-of-biomachinery-can-VyP0pIvJSCCQdtbQEQAqZg.md
27. bioscale-programming-integrate-d4MXJYjQQpmh.sA28C.GUg.md
28. bioscale-tech-inc-is-a-hardwar-NaUHAgpQSBSDMt3OtEEQWA.md
29. bioscale-tech-inc-instructions-d4MXJYjQQpmh.sA28C.GUg.md
30. what-can-be-discovered-to-help-IOXF759yT2WQkMCzzEMxXA.md
31. the-github-csv-renderer-fixes-bJt8YZqsQC2IQxznEETIiw.md
32. you-can-treat-your-new-neuron-iLnthLBcTTq9vsCNpGWejA.md
33. what-can-we-create-in-new-synt-2WCDckpPQ4WMwPXcjj55vA.md
34. daily-adjacent-domain-research-VcJN7nsDTI.iZ0yZpBfXYQ.md
35. organically-integrated-augment-Dz2V_eZ9QHyTACOSR97Pzw.md