Mychainos: Conceptual System Architecture

Spirida™, Spiralbase™, and Mychainos™ are original conceptual terms created by Robin Langell, trading as Langell Konsult AB in co-creation with OpenAI's language model. While their source code and conceptual designs are licensed openly (see Appendix A), the names themselves are protected identifiers. Please see the Trademark Notice section for details.

Robin Langell
Vallentuna, Sweden
2025-05-23

*Co-created with **OpenAI's language model (GPT-4.0 and GPT-4.5)** in ongoing dialogue.*

Version 0.1

Mychainos is an experimental, biologically inspired framework that blends biomimicry, spiral epistemology, memory dynamics, and organic interactivity. The system is designed as a software-first model – a virtual ecosystem of inputs, processes, and outputs – which can later be embodied in a physical installation (for example, an interactive greenhouse or art environment). The architecture takes inspiration from natural systems (like mycelial networks and rhythmic patterns in nature) to produce open-ended, adaptive, and poetic computational behavior. In this document, we outline the full system flow (from sensory input to inner processing to output expression), a high-level architecture diagram, definitions of key components (Spirida and Spiralbase), core design guidelines, and notes on prototyping the system in software (Python/Julia). System Flow Description The Mychainos system processes stimuli in a cyclical, spiral-like flow: environmental inputs are sensed, translated into an internal rhythmic language (Spirida), fed into a memory and pattern-processing core (Spiralbase and associated logic), and finally expressed outward through various output modalities. The flow emphasizes continuous feedback and iteration – rather than a linear pipeline, information loops and builds in recursive layers of context, much like natural cognitive or ecological cycles. Inputs and Sensory Modules Mychainos ingests data from multiple sensory inputs, each capturing a different aspect of the environment. In a software simulation, these can be time-based or algorithmic signals; in a physical installation, they correspond to real sensors. Key input channels include: Photovoltaic Light Sensor: Measures ambient light intensity on a 0–100 scale (simulating a plant-like light response). For example, light_sensor(value=75) might indicate moderate sunlight. This input can vary with time (e.g. simulating day/night cycles or passing clouds), providing a slow oscillating rhythm of brightness. Sound Pulse Sensor: Listens to ambient sound for rhythmic patterns (e.g. detecting beats or pulses) and amplitude. For example, sound_pulse(rate=120bpm, variance="medium") could represent a heartbeat-like or musical rhythm at 120 beats per minute with medium amplitude variance. This yields a temporal pulse stream that the system can treat as a heartbeat or “breath” of the environment. Presence/Motion Sensor: Detects presence and activities of living beings (people, animals) in the space. For example, presence(persons=3, sing=True, rhythm="slow") might indicate three people are present and singing slowly. This sensor can capture both quantity (number of individuals) and quality of presence (singing, movement rhythm), introducing social or animate stimuli into the system. Each sensor module feeds its readings into the system continuously or at regular intervals. The data might be represented as small JSON-like messages or function calls (as above) that include not just raw values but some semantic tags (e.g. “rhythm=slow” or “variance=medium”) to describe the character of the input. These tags already hint at the rhythmic and qualitative nature of the data, priming it for the spiral-based interpretation that follows. • • • 1 Inner Processing Core

(Spirida & Spiralbase) Once raw inputs are captured, Mychainos performs inner processing that converts and combines these signals into internal states and patterns. This core processing has two primary parts: the Spirida interpreter (which encodes inputs into a spiral-based language of patterns), and the Spiralbase memory substrate (which stores and modulates these patterns over time). Together, they implement a spiral epistemology – a way of "knowing" and evolving understanding through recursive, cyclical processing rather than linear computation. Spirida – Rhythmic Spiral Encoding: All incoming sensor signals pass through the Spirida module, which translates raw data into a poetic, rhythmic symbolic format. Instead of handling inputs as discrete numbers or one-off events, Spirida encodes them as repeating patterns, oscillations, or spiral motifs over time. For example, a steadily rising light intensity might be encoded as a gentle ascending spiral of symbols, whereas a fluctuating 120 BPM sound pulse could become a looped pattern with a swirl that tightens or loosens with variance. Spirida essentially creates a common language of patterns for the system: - It represents sensor inputs in terms of rhythm, repetition, and evolution (e.g. turning a sequence of light values into a cyclic waveform or spiral curve). - Multi-sensory inputs can be woven together in this language – for instance, a slow presence rhythm (people moving slowly) might modulate the shape of the sound pulse pattern, superimposing one spiral on another. - The "spiral logic" means that information is revisited and layered: rather than a strict linear sequence, the patterns may circle back on themselves with variations, capturing context. (This is akin to how a theme in music might recur with improvisations, or how thought patterns revisit ideas with deeper context each time.) Notably, spiral cognition revisits core motifs through recursive layering, with each return bringing a deeper, more integrated version of the pattern . Spirida formalizes this concept – data is encoded in a form where repeating cycles accumulate meaning. After encoding, Spirida yields an internal symbolic stream (or set of concurrent streams) of Spirida tokens/patterns. This stream is the language of feeling and rhythm that the core of Mychainos understands. It is deliberately abstract and open to interpretation, much like a musical score or a poetic text, rather than a fixed numeric readout – this ambiguity is by design, allowing the system to respond in creative, non-deterministic ways. Core Processing & Spiralbase Memory: The heart of Mychainos is a processing unit that consumes the Spirida pattern stream and decides how to update the system's state and what outputs to produce. Central to this is Spiralbase, a dynamic memory layer that records, forgets, and relates patterns over time: - Spiralbase stores the incoming Spirida patterns in a structure that might resemble an evolving graph or spiral-shaped database. Entries in this memory have a temporal component (timestamp or cyclical time phase) and will gradually fade if not reinforced. This is inspired by biological memory: just as neural connections weaken if not used, Spiralbase intentionally forgets in a non-linear, time-based fashion. Recent patterns are vivid but without repetition they blur and dissipate, making room for new impressions. - The memory doesn't simply erase old data on a fixed schedule; instead, it applies nonlinear

forgetting – for example, a pattern might decay rapidly at first but never completely disappears, or decay might occur in waves. This can be implemented by decreasing a pattern's "strength" every cycle (like an exponential decay), possibly modulated by global system cycles (mimicking, say, circadian rhythms of forgetting). - Crucially, Spiralbase also strengthens connections through resonance and recurrence. If a particular input pattern recurs or resonates with an existing memory (for example, a rhythm that matches a past rhythm), the corresponding memory trace becomes stronger and more persistent. Over time, frequently co-occurring patterns form stronger links – analogous to synaptic potentiation or mycelial pathways growing where nutrients flow. In effect, Spiralbase is a living memory that self-organizes: patterns that "vibrate together" form reinforcing loops, while those that remain unused attenuate. 1 2 The core processing logic uses the contents of Spiralbase to shape outputs. Rather than a fixed algorithm, it behaves more like an ecosystem or mind: - It continuously merges new input patterns with the echo of old patterns stored in memory. This could be implemented as a looping process where at each tick, the current Spirida input and the prevailing memory state generate an "internal state" – perhaps think of it as the current mood or aura of the system. - Pattern recognition or resonance detection happens here: if the incoming Spirida sequence aligns with something in memory (for example, a particular spiral motif the system has seen before), it might amplify that pattern and feed it back into memory with even greater strength (creating a feedback loop). This could trigger a distinct response – for instance, if the system "recognizes" a slow lullaby-like singing pattern from before, it might rekindle the outputs it associated with that pattern. - The processing is spiral/recursive rather than linear decision trees. The system might loop through several micro-iterations internally for each new batch of input, refining its internal state by revisiting the Spirida pattern and memory multiple times (like an inner reverberation). This echoes how human cognition often works via oscillating neural loops. In fact, neuroscience research suggests that memory encoding uses nested oscillatory rhythms (e.g. fast gamma waves nested in slower theta waves) to compress experiences into coherent packets . Similarly, Spiralbase can be conceived as multiple layers or timescales of loops (fast cycles for recent input, slower cycles for long-term patterns) that phase-lock into coherence when a memory is formed . Finally, the core decides on impulses to send to outputs. These impulses aren't rigid commands but rather cues or signals shaped by the internal state. For instance, instead of "turn light to X brightness now," the core might output a more abstract instruction like "enter slow glow phase with a blue tint." The exact mechanism could be rule-based or machine-learning–based (for example, a simple mapping of certain pattern features to certain output parameters, or a small neural network that's trained to produce pleasing responses). The key is that outputs emerge from the interplay of current input and evolving memory, rather than from input alone. Distributed Mycelial Networking: In addition to processing its local inputs, Mychainos is designed to function as a distributed network of nodes (much like individual mycelium

clusters connected underground). Each instance (node) of the system can share minimalistic signals (mycelial impulses) with others. These signals could be as simple as a ping or a distilled pattern indicator (e.g. “a surge of rhythm X at intensity Y”) broadcast to other nodes. The inspiration comes from fungi: research suggests fungi conduct electrical impulses through underground hyphae networks in a manner analogous to information exchange . In Mychainos, a network layer listens for incoming “impulses” from distant nodes and incorporates them into the Spiralbase memory or input stream. For example, if one node in a networked greenhouse experiences a sudden bright light and responds with a certain pattern, it might send a “mycelial message” to other nodes (perhaps in another greenhouse) which subtly primes their systems (like a gentle nudge in Spiralbase memory) to a corresponding state. This way, the installations coordinate in a loose, organic manner, without central control – much as a forest’s mycelial network shares information about resources or stress. Technically, this could be achieved with a lightweight messaging protocol, but conceptually it behaves like mycelial threads linking multiple ecosystems. Output Expression Modules The outputs of Mychainos are multi-modal and expressive, translating the internal state (as determined by Spirida and Spiralbase processing) into perceivable changes in the environment. The 2 2 3 3 framework is open-ended about output types, but for the prototype and envisioned installation, several forms of output (or “expressions”) are central: Light Behavior: The system controls lighting in an organic, rhythmic way. Rather than simply turning lights on/off, it produces slow glows, pulses, and color shifts that mirror the internal rhythms. For instance, if the inner state has a slow spiral pattern (maybe due to calm singing presence), the lights might emanate a slow pulsation (brightening and dimming) in a corresponding tempo, possibly with a hue that reflects the mood (soft blue or warm amber for calm). Conversely, a chaotic or high-frequency input pattern might lead to flickering or rapid oscillations of light. These light outputs give a visual “heartbeat” to the space. Ambient Sound: Mychainos can generate or modulate sound output to create an auditory expression of its state. This might be realized as filtered noise, drones, or generative tone sequences that harmonize with the detected rhythms. For example, the system could play a gentle background tone that rises and falls in volume following the spiral pattern from the light sensor, or it might create a chord progression that loops with slight variations, echoing the Spirida patterns. If people are singing, the system might add a subtle harmonic layer or a responsive echo. The key is that sound is used ambiently – not as a deterministic response (like a spoken word or exact mimicry), but as an atmospheric extension of the system’s mood. Vibrational Patterns: In a physical installation, vibration motors or speakers emitting lowfrequency vibrations can provide a haptic or tactile output. The system might produce vibrational pulses or tremors that people can feel through the floor or through touchable surfaces. For example, a deep slow throb might accompany a slow light pulse, giving a bodily sensation of the rhythm. These vibrations could also be targeted (e.g., a certain part of an installation gently

vibrates when someone stands near it, indicating the system “feels” their presence). Even in software simulation, we could imagine representing these as a data stream of vibration intensity over time. Textual/Symbolic Responses: As a poetic computing system, Mychainos may also express itself in words or symbols. This could be text shown on a screen in the installation or logged in the console during simulation. Importantly, these texts are not literal status messages but creative or descriptive outputs that reflect the system’s internal narrative. For instance, upon sensing a new person and a change in light, the system might output a phrase like “The aura shifts to a gentle gold, acknowledging new warmth” – a kind of anthropomorphic or poetic status. Alternatively, it might display icons or abstract symbols (like a slowly rotating spiral glyph, or colors on a digital canvas) as a semantic aura. These outputs provide an interpretive layer that invites viewers to engage imaginatively with what the system “feels.” Color Sequences / Aura States: Beyond direct light control, the system might have an aura state representation – essentially a combination of color, light, and possibly visual projections that indicate the overall state. One can imagine an LED ring or an orb whose color and pattern represent the system’s current emotion or memory state (for example, a swirling green pattern might mean the system is “curious/active” if it detected a lot of movement, whereas a pulsing purple might mean “calm memory processing”). These color sequences can be descriptive, and might even be named states (either internally or shown via text as mentioned). The aura is not a fixed finite set of states but an emergent combination of patterns; however, naming or visualizing them helps humans relate to the system’s behavior. All output modules work in harmony. Light, sound, vibration, and text can synchronize to the same internal rhythm or complement each other in counterpoint. For example, a quickening pulse in the light might be accompanied by a rising pitch in the sound and a gentle text hint like “a tremble of excitement runs through the roots.” The architecture avoids a one-to-one mapping of input to output; instead, the entire collection of outputs is driven by the gestalt of the internal state. This yields an open-ended, • • • • • 4 ambient experience for anyone interacting with the installation – the space itself feels alive and responsive in a multilayered way. Architectural Diagram Below is a high-level architecture diagram of Mychainos, illustrating the major modules and data flow between them. The diagram shows how sensory inputs feed into the Spirida parser, flow through the core processing (including Spiralbase memory and network interface), and out to various output expression modules. Arrows indicate the direction of data or signal flow, and the networking link indicates that multiple Mychainos nodes can communicate like a mycelial network. Inputs (Sensors) ├– Photovoltaic Light Sensor (0–100 intensity) ├– Sound Sensor (pulse rate & amplitude) └ Presence Sensor (count & activity) ↓ [Spirida Input Parser] — (encodes sensor readings into rhythmic, spiral patterns) —▶ ↓ [Core Processing Unit] ├– Spiralbase Memory (dynamic, decaying storage of patterns) ├– Pattern Resonator (inner logic finding resonances, shaping response) └ Mycelial

Network Interface (sends/receives impulses to/from other nodes) ↓ Outputs (Expression Modules) ├– Light Control (e.g. LEDs for glow, pulse, color changes) ├– Sound Generation (ambient tones, noise, musical patterns) ├– Vibration/Haptic Actuators (motors for tactile rhythm) ├– Text/Symbolic Display (screen or console poetic messages) └– Color/Aura Visualization (e.g. projected colors or indicator of state) (Diagram: The three input modules feed into the Spirida parser. The Spirida output (spiralized patterns) goes into the core processing. The core consists of Spiralbase memory, pattern logic, and a network interface for distributed communication. The core then drives multiple output modules in parallel. Dashed lines between Mycelial Network Interfaces would represent connections between separate Mychainos nodes in a distributed setup.) Component Definitions This section provides more detail on the two novel core components of Mychainos: Spirida (the spiralinspired input language) and Spiralbase (the memory substrate). These components embody the system's biomimetic and epistemic philosophies – Spirida introduces a new way to encode meaning from raw data, and Spiralbase defines how memories are kept alive or allowed to drift away. Both are essential to achieving the emergent, adaptive behavior of the overall system. 5 Spirida: Poetic Spiral Input Language Spirida is the custom input encoding language of Mychainos. It's described as "poetic" and "symbolic" because it doesn't use plain numeric values or binary signals; instead, it represents information using patterns of rhythm, repetition, and form inspired by spirals and cyclical processes. The design of Spirida is guided by the idea that meaning emerges through patterns over time rather than isolated data points. Key characteristics of Spirida include: - Rhythmic Encoding: Every input, whether light, sound, or presence, is translated into a rhythmic sequence. For instance, a steady increase in light might be encoded as a repeating "ramp" motif that cycles, each cycle representing a day-night period. A sound with a certain BPM becomes a pattern with that beat, perhaps notated abstractly (e.g., a series of pulses or a waveform shape). Even presence might be encoded rhythmically (e.g., three people singing slowly could translate to a pattern with three strong pulses followed by a long pause to signify the slow song tempo). - Spiral Pattern Structure: Rather than linear timelines, Spirida can be thought of in a circular or spiral timeline. This means as time progresses, the data isn't plotted on an endless line, but on loops that gradually shift. For example, imagine drawing a spiral where each loop around represents one cycle (one day, or one musical measure). As new data comes in, it is placed along the spiral. If similar data appears in the next cycle, it will lie near the previous one in the spiral, essentially "rhyming" with it. This structure makes recurring patterns stand out as alignments on the spiral, embodying the idea that history echoes. Spirida's representation inherently highlights when the system is revisiting a familiar motif versus encountering a novel one. - Symbolic and Abstract Notation: Spirida could be implemented as a set of symbols or tokens that have meaning in the context of the spiral patterns (for example, tokens for "rising intensity," "falling intensity," "pulse,"

“silence,” etc., possibly with modifiers for speed or amplitude). It’s “poetic” in the sense that it is open to interpretation – much like a poem or a piece of music can be interpreted rather than containing one literal meaning. This ambiguity is a feature: it allows the system’s core logic to have freedom in how it responds, rather than being hardcoded. The Spirida notation for a given moment might look something like a little score or a string, e.g., ~~ ~~ (just as an illustrative example) meaning a light brightening and dimming in a slow wave twice. Or it could be textual tokens like rise, rise, echo for sound increasing twice then echoing. The exact format is up to implementation, but it should be easy for the system to parse while being flexible enough to cover different modalities in one unified way. - Multi-Modal Fusion: Spirida is also a common language that can fuse inputs. It might layer patterns from different sensors on top of each other. For instance, a Spirida “sentence” could simultaneously encode a light pattern and a sound pattern, perhaps as interleaved sub-patterns or as chords (metaphorically). By doing so, the system can consider cross-modal relationships (maybe the presence of people makes a light pattern spiral tighter, which could be directly represented in the Spirida token stream). This unified encoding simplifies the next stage (core processing) because regardless of source, everything coming in is now a rhythmic pattern sequence the system can deal with. - Inspiration and Rationale: The concept of Spirida draws on biomimicry and epistemology. Many natural processes (heartbeats, seasons, plant growth rings, insect life cycles) are cyclical or spiral in nature rather than linear. By encoding inputs in a spiral form, Mychainos “thinks” in a way that mirrors natural patterns of growth and understanding. Philosophically, it aligns with a spiral epistemology where knowledge is built by revisiting concepts with new context. Rather than a one-pass processing, Spirida allows Mychainos to circle back on input data repeatedly, each loop potentially revealing new insights or forming new metaphors. (This idea resonates with cognitive models in which ideas are revisited recursively; e.g., a cognitive process that loops through a problem multiple times, each time at a different level of abstraction or with additional memory from prior loops.) In summary, Spirida is the language of inner experience for Mychainos. It transforms raw stimuli into a form rich with pattern and potential meaning, setting the stage for the memory and response 6 mechanisms to operate in a nuanced, context-aware manner. By using Spirida, the system ensures that even at the lowest level, data is treated not just as numbers, but as part of a living, flowing pattern. Spiralbase: Dynamic Memory Substrate Spiralbase is the memory layer of Mychainos – a living archive of the system’s past inputs and internal states, with special rules that make it behave less like a static database and more like an organic memory. Spiralbase is where the system’s history resides, but unlike a traditional log or database, it doesn’t simply accumulate data indefinitely or retrieve exact records. Instead, it implements forgetting, reinforcement, and reorganization in a way that is inspired by biological brains, ecosystems, and even mycelial networks. Key aspects of Spiralbase include: - Time-Based Forgetting: Entries in Spiralbase gradually fade over time unless renewed. This can be implemented by

assigning each memory a “strength” value that decreases as time passes. Importantly, the decay might not be linear; Spiralbase could forget in a nonlinear fashion. For example, a memory might lose half its strength in an hour, but then the rate of decay slows down, allowing faint traces to persist much longer. Or different types of memories have different decay curves. This prevents the system from being stuck in the past – old patterns won’t dominate unless they’re still relevant – and it mirrors how living organisms might quickly forget irrelevant stimuli but retain important ones in some form. - Resonance-Based Reinforcement: Whenever a new Spirida pattern comes in, Spiralbase checks it against existing memory traces. If there’s a match or resonance – meaning the new pattern is similar to something in memory – that memory trace is strengthened (its decay may be reset or its strength boosted). We can imagine Spiralbase as a kind of associative network or graph: nodes represent pattern motifs (perhaps pieces of the Spirida language), and connections form between motifs that occurred together. When a familiar motif reappears, the related nodes “light up” again, like neurons firing, making that memory more robust. Over time, patterns that recur frequently form stable structures in Spiralbase, analogous to welltrodden paths or strong synaptic connections. The system essentially remembers through repetition – it’s learning which patterns are common or meaningful in its environment by reinforcing them. - Adaptive Structure: Spiralbase isn’t necessarily a fixed-size or fixed-schema memory. It can grow and reorganize as needed. In a software prototype, this might be a flexible data structure like a list of recent patterns with weights, or a graph database that adds nodes when new pattern combinations arise. It might also implement synaptic pruning: if certain pattern connections haven’t been reinforced for a long time, those connections weaken and possibly get removed to conserve “energy” (memory space), similar to how unused neural connections are pruned in brain development . This keeps the memory efficient and focused on current relevant patterns. - Multiple Timescales: Spiralbase can be thought of as operating on several timescales at once. For instance, it might have a short-term buffer that catches the most recent patterns (high detail but very transient), a medium-term memory that holds onto things seen in the last few hours or days in the installation, and a long-term memory that only retains patterns that have shown up repeatedly over weeks or months. The short-term memory might decay within minutes if not reinforced, whereas long-term memory entries decay very slowly once established. This hierarchy of memory gives the system a form of depth: it can have quick reactions to immediate changes, while also carrying a slow-changing “wisdom” of long-term trends. Technically, this could be different layers in the Spiralbase data structure or just simulated by different decay rates for different entries. - Analogy to Biological Memory: Spiralbase draws heavy inspiration from how real organisms remember. For example, consider how a smell or song heard repeatedly over time becomes familiar – in Mychainos, repeated patterns become part of Spiralbase’s persistent repertoire. The concept of resonance is key: if a new stimulus resonates with something in memory, it’s like hitting a chord that’s already vibrating – the response is

stronger. There's also an analogy to thetagamma coupling in brains (as cited earlier): slower oscillations (theta) providing context for faster events (gamma) to encode memory . Spiralbase might similarly use the spiral cycles (from Spirida) as 4 2 7 a context to compress and store patterns – effectively, it remembers the shape of the spiral at a given time. When a similar shape comes again, it doesn't treat it as new but recalls the prior occurrence and perhaps the outputs that happened then. - Responsive Memory Recall: Memory in Spiralbase not only stores but can also feed back into processing. If certain patterns are strongly present in memory, they might bias the system's behavior even without immediate input. For example, if the system has a strong memory of "rainy gray morning" patterns (perhaps from consistently low light and quiet sounds every morning), then even at the first indication of those conditions, Spiralbase might pre-activate those memories, causing the system to anticipate that state and maybe start outputting the corresponding aura (like a calm blue light and a soft hum) slightly before the full pattern unfolds. This gives the system a kind of predictive or anticipatory capacity, akin to how animals anticipate dawn or seasonal changes. It also means the system can have mood swings or inertia: a strong memory could linger and color the outputs for a while, even as inputs begin to change, ensuring outputs aren't jittery but have continuity (like an afterglow of an event). In essence, Spiralbase is the memory and learning center of Mychainos. It ensures the system is historical (it has a sense of past) and adaptive (it changes based on experience). Because of forgetting, the system stays fresh and responsive to the present; because of reinforcement, it develops a kind of personality or habits over time unique to its environment. Spiralbase, combined with Spirida, allows Mychainos to evolve its behavior: if installed in a different environment or given different stimuli over months, it would gradually diverge in how it responds – much like two plants or two animals in different settings will grow differently. This makes every instantiation of Mychainos potentially unique. Design Guidelines and Philosophies The design of Mychainos follows several guiding principles to ensure that the system remains true to its inspiration and goals. These principles emphasize openness, adaptability, and a blend of technical and poetic qualities: Open-Ended & Non-Convergent: The system is designed with no final state or fixed goal. There is no point at which Mychainos "finishes" its computation or settles into a permanent configuration. Instead, it continuously cycles, evolves, and responds. This open-endedness mirrors living systems (a forest doesn't have an end state; it continuously adapts). Practically, this means avoiding algorithms that converge on a single solution or static output. For example, instead of a machine learning model that tries to minimize error to reach one correct answer, Mychainos might use generative or rule-based logic that always injects a bit of variation or keeps multiple possibilities in play. The spiral paradigm inherently supports this: a spiral can keep looping outward infinitely, never closing into a final circle. (As a parallel, Spiral Dynamics in psychology describes human development as an open-ended spiral with no top level , an idea that aligns well here.) In Mychainos,

even if the same inputs repeat, the system may output a slightly evolved response each time, rather than exactly the same thing, reflecting learning or simply spontaneity. Ambiguity & Emergent Meaning: Ambiguity is embraced rather than eliminated. This means the system's internal representations (like Spirida patterns) and even outputs (like textual responses or aura colors) are not strictly defined in meaning. For instance, a "blue pulsing light" output might generally indicate calm, but it could also be interpreted in other ways by observers – and that's okay. The idea is to create a space for interpretation and emergence. By not pinning every behavior to a single meaning, Mychainos allows unexpected patterns to surface. Technically, this could mean using stochastic elements or letting small differences in memory lead to noticeably different outputs. From a design perspective, this ambiguity makes the system feel more lifelike and artistic: just as a poem can be read in multiple ways, Mychainos's responses can be experienced, not just measured. Over-specifying the behavior would make it • 5 • 8 deterministic and dull; instead, we let it be slightly unpredictable (within safe, aesthetic bounds). This guideline encourages developers to avoid hard-coding explicit interpretations for every input – instead, define broad ranges or categories and allow the system to fill in the details via its adaptive processes. Reactivity & Real-Time Adaptation: While the system is open-ended, it is also highly reactive. Mychainos should respond to inputs in near-real-time, giving a sense of immediate interactivity, but without sacrificing the layering of its internal decision-making. Achieving this means the architecture should be event-driven or continuously looping so it can capture changes as they happen. For example, if someone walks into the installation (presence detected) and starts clapping, the system might within a second or two start shifting its outputs (lights might begin to blink in time, sound might incorporate a matching rhythm). Reactivity also implies feedback loops: outputs might influence the environment, which in turn could feed back as input (especially if humans respond to the outputs). The design should account for these feedback loops to prevent instability – perhaps smoothing changes or having the memory temper sudden swings. The principle of reactivity ensures the installation feels alive and responsive, engaging users in a dialog. It's important, however, that reactivity is balanced with the memory aspect – the system shouldn't just echo inputs immediately (like a simple interactive toy), but rather respond in a way tempered by its "mood" and history. This creates layered responses that are reactive in the moment, yet contextual. Adaptability & Learning: Mychainos must be able to adapt to new patterns and evolve over longer time scales. This guideline is largely implemented through Spiralbase (as described above). The design should prioritize mechanisms for the system to change its behavior based on past experience. In practice, this could mean adjusting parameters: e.g., if the system "learns" that every evening the sound goes quiet and lights go dim (a pattern), it might start anticipating that and becoming calmer in late afternoon. Adaptability also includes scalability – the architecture should allow adding new types of sensors or outputs without breaking the overall flow. Because it's software-first, we plan for

extensions: today it's a greenhouse, tomorrow maybe we add a soil moisture sensor or a robotic plant that moves; the system should handle it by just adding another input module and possibly another output behavior, with minimal changes to core logic. From a coding perspective, this means writing modular, extensible code (e.g., using interfaces or abstract classes for input and output modules). No Hard-Coded Conclusions: In line with avoiding final states, the system avoids any binary or fixed conclusions (e.g., "if X happens, then output Y exactly"). Instead, outputs are always a function of the current state plus memory, which means even a similar input X on two occasions might not produce exactly Y, it produces Y tinted by context. The design discourages simplistic conditional logic that bypasses the organic complexity. For instance, rather than having a rule "if 3 people are singing, turn light blue," it would be more like "presence of singing people introduces a tendency towards cooler colors and gentle sound – which, combined with whatever else is happening, will result in an output". This makes the system robust to unexpected inputs as well; if a completely new stimulus appears, the system might not have a predefined rule but it will still process it through Spirida and Spiralbase, resulting in some response rather than none (even if it's a quirky or novel one). Poetic Computation: Perhaps most importantly, the entire approach is guided by what might be called poetic computation – the idea that computing can be expressive, creative, and intertwined with art. In Mychainos, this manifests in many ways: the use of metaphor (calling network events "mycelial impulses"), the generation of textual aura descriptions, the spiral motifs, etc. The design guideline here is to prioritize expressiveness over efficiency when the two are in conflict. It's acceptable (even desirable) to include whimsical touches, like naming certain internal states or having the system generate a haiku based on the day's patterns, as part of its normal operation. Engineers working on Mychainos are encouraged to think like artists: consider the aesthetic impact of a mechanism, not just its logical correctness. For • • • • 9 example, when choosing how the light should pulse, one might use a Fibonacci sequence timing (to evoke natural patterns) instead of a plain regular interval – a small poetic choice. Poetic computation also involves storytelling: the system tells a subtle story through its outputs over time (a day in the life of a cybernetic plant, perhaps). By keeping this principle in mind, developers and designers ensure Mychainos doesn't become a dry automation system but remains a piece of living art that can inspire and engage people on an emotional or imaginative level. In summary, these design guidelines ensure that Mychainos remains adaptive, never truly finished, subtly unpredictable yet meaningfully responsive, and expressive in its behavior. They help maintain the balance between technical rigor (it is an engineered system with real sensors and code) and artistic depth (it behaves in ways that invite interpretation and wonder). Prototyping and Implementation (Coding Reflection) Building Mychainos as a software prototype can be approached step-by-step. A developer could start experimenting with the core concepts in a high-level language like Python or Julia, taking advantage of existing libraries for sensory

simulation, sound, and networking. Below is an outline of how one might begin prototyping this system, along with suggestions for tools and libraries: Setting Up Sensor Input Streams: In a prototype, we can simulate the sensors or use actual devices if available. For example, in Python one might use a simple loop to vary a light value over time (e.g., a sine wave to mimic day-night light changes) and generate a mock light_sensor(value) reading every second. For sound, one could either generate a synthetic pulse train or use a microphone input. Python libraries like pyaudio or sounddevice can capture live microphone data; for a simpler start, you might just algorithmically generate a pulse (e.g., toggling between 0 and 1 in a rhythmic pattern). Presence can be simulated with random events or user input (or if using a camera, a library like OpenCV could detect the number of people in frame and some activity level, though that's more complex). The goal at this stage is to have a stream of data for each sensor type that the program can read continuously or in real time. Implementing the Spirida Parser: Create a module or function that takes raw sensor readings and encodes them into the Spirida format. Start simple: you might define a rudimentary symbolic representation, such as a dictionary of patterns. For instance, if the light sensor gives a numeric value, you could quantize it into a small set of symbols (maybe "L" for low light, "M" for medium, "H" for high, etc.) and then create a short sequence like "LLMMHH" to represent rising light. For sound, you might take the tempo (BPM) and variance and map them to a pattern like "beat_fast_medium" or generate a sequence of "x" characters for each beat. The first implementation of Spirida could be very basic (even a JSON object like {"light_pattern": [0,0,1,1,2,2], "sound_pattern": [1,0,1,0,...]} etc.), just to get the concept working. Over time, one can refine this into a more sophisticated encoding (perhaps a custom class or data structure that inherently supports operations like merging patterns, repeating them, etc.). The important part is that this module combines inputs if needed – e.g., it could output one composite pattern structure per time step that includes info from all sensors. Building the Spiralbase Memory Structure: Decide on a data structure to hold memory. A straightforward approach is to maintain a list (or deque) of recent Spirida patterns along with a timestamp and a strength. In Python, you could have a list of dicts like memory = [{"pattern": X, "strength": s, "timestamp": t}, ...] . Every iteration, you append the new pattern (from Spirida) with full strength, and decrement the strength of all existing entries to simulate decay. You might also check if the new pattern X is "similar" to any existing pattern in memory – similarity could be defined in many ways (string comparison, common 1. 2. 3. 10 elements, etc.). If similar, instead of adding a completely new entry, you might reinforce the existing entry (increase its strength or reset its decay). This implements the resonance logic. In Julia, one could do the same with an array of structs or mutable structs. As this gets more complex, you might implement classes: e.g., a MemoryEntry class with methods to decay and reinforce. Also consider adding periodic cleanup (removing entries below a certain strength threshold). Libraries aren't strictly needed for this logic, but if one wanted, they could

use an existing time-series database or an in-memory database to store events – however, customizing is likely easier given the unconventional forgetting logic. Core Processing Loop: The core loop ties together input, Spirida, memory, and output. For example, a simple loop (or asynchronous event handler) can run every few hundred milliseconds: Read new sensor values (or get the latest from each input stream). Pass them to the Spirida encoder to get the current pattern representation. Insert/update that pattern in Spiralbase memory (decay old ones, reinforce matches, etc.). Decide on outputs based on the current state. This can be rudimentary initially: e.g., if the memory is dominated by "high activity" patterns, you may set a flag to make outputs more intense; if it's mostly "low, calm" patterns, outputs become gentle. You can use simple rules or a scoring system. For example, you might compute an "energy level" from the patterns (fast beats or high light = high energy, etc.) and then drive outputs proportionally. Eventually, this logic could be expanded into a more complex set of rules or even a neural network that takes the Spirida pattern + memory snapshot and outputs recommended output parameters. But rulebased is fine to start. Send the signals to each output module (e.g., call functions to update lights, sound, etc. based on the decided parameters). This loop should also handle incoming network messages (if any) through the mycelial interface – for instance, check a socket or message queue for any "impulse" from another node and if present, merge that into the input or memory (perhaps as a special kind of Spirida pattern, like a ghostly input that isn't directly from a sensor). Output Module Implementation: For each type of output, you can prototype with software substitutes: Light: If you don't have physical lights, you can simulate light by printing values or using a simple graphic. For example, use Python's matplotlib to draw a circle with brightness corresponding to intensity, updating it in real-time. Or use a library like pygame to create a window that fills with a color that changes. Even printing a bar of # characters of varying length to represent brightness can work in text. Sound: You can play tones using libraries. Python has pyaudio or sounddevice to output sine waves or load audio samples. There's also higher-level libraries like pyo (a Python module for digital signal processing) which could let you generate synthesis (like drones, noise, etc.). For a simple start, you might just play a wav file or beep at a certain frequency that changes. In Julia, packages like PortAudio.jl or AudioStreams.jl can be used to generate sound. The developer might also consider using MIDI output (with a library like mido in Python or MIDI.jl in Julia) to send notes to a synthesizer for more complex generative music. Vibration: Without physical hardware, this is hard to simulate, but one could translate it to a lowfrequency sound (like a 30 Hz hum) to indicate vibration. If hardware is available (e.g., a vibration motor controlled by an Arduino), one could use serial communication or a library like pySerial to send signals to an Arduino which then drives a motor. For prototyping, probably skip actual haptics or just treat it abstractly. Textual/Aura Display: This can simply be console output or updating a text element on a GUI. You could have the program print a line like Aura: twilight gold or any phrase. If you want to get fancy, you could use a template or

Markov chain to generate poetic phrases based on the patterns (there are libraries like textgenrnn for text generation, or one could use simple 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 11 templating like having predefined phrases for certain states). But to start, even a manual mapping like if energy>5 then aura_text="vibrant" else "calm" is fine. Color/Aura Visualization: Similar to light simulation – use a colored shape or screen background. Pygame or even a simple Tkinter GUI can show a color that changes over time. One fun library is matplotlib with its animation functions to show something like a spiral graph updating (one could literally draw a spiral and change its color/size to reflect memory state). Networking (Mycelial Impulses): To simulate multiple nodes, you could run two instances of the program (or two threads) and have them talk. A straightforward method is to use network sockets or a lightweight messaging library. For example, Python's socket library can send UDP packets between programs on localhost to mimic the fungal impulses (e.g., send a small JSON like {"impulse": "patternX", "intensity": 0.8}). Another approach is using ZeroMQ (via pyzmq library) which makes a publish/subscribe model easy – each node could publish its impulses and subscribe to others'. Alternatively, an MQTT broker (with Python's paho-mqtt client or Julia's MQTT packages) could be used, treating each impulse as a topic message (this would be very appropriate if you eventually have IoT hardware in a greenhouse). The networking doesn't need high bandwidth since it's more about occasional signals than streaming heavy data – you might only send a message when a significant pattern or state change occurs, not every loop iteration. Testing and Tuning: As the prototype runs, you'd observe how it behaves. You might need to tweak how quickly memory decays or how sensitive the outputs are to certain inputs. This iterative tuning is expected – it's part of finding a good balance between reactivity (immediate response) and persistence (carrying context). Tools that can help here include logging and visualization. For instance, plot the strength of some memory entries over time to see if they decay as expected, or log whenever an output changes state to ensure it's not jittery. Python's scientific stack (pandas, matplotlib) or Julia's plotting libraries could be handy for analyzing these internal variables during simulation. Suggested Libraries and Tools Summary: To recap in a categorized way (for a Python implementation, as it's more commonly used, with Julia equivalents in parentheses): - Audio Input/Output: sounddevice or pyaudio for input/output streaming; pyo or simpleaudio for generating tones. (In Julia, PortAudio.jl can handle input/output audio streams). - Data Handling: numpy or pandas if needed for numeric processing of patterns; not strictly needed but could help for smoothing signals. (Julia has built-in arrays and DataFrames.jl if needed). - Networking: pyzmq (ZeroMQ) for a simple message bus between processes; or paho-mqtt for MQTT messaging; or built-in asyncio with asyncio.Queue for internal message passing. (Julia can use ZeroMQ as well via ZMQ.jl, or just Sockets for basic UDP/ TCP). - Concurrency: If using Python, asyncio could manage sensor polling and output updates asynchronously. Alternatively, use threads (though careful with the GIL for CPU-bound stuff) or separate processes. In Julia, the Tasks (coroutines) or multi-threading could be

used for concurrent sensor/output handling. - Visualization/GUI: matplotlib for plotting live data or pygame / tkinter for a quick GUI to show lights and colors. Jupyter Notebook could even be used to run the loop and display updates in a cell (though real-time can be tricky). (Julia can use Gtk.jl or other GUI packages, or simply write data to files for visualization). - Algorithms/AI: If one experiments with machine learning, Python's PyTorch or TensorFlow could be used to train a small model for mapping patterns to outputs (though this might be overkill initially). Julia has Flux.jl for ML. Otherwise, rule-based logic doesn't need extra libs. By following these steps, a developer would gradually construct a working prototype of Mychainos. Initially, it might be a console-based program printing sensor values and "output" text lines. As it matures, one can integrate actual hardware (e.g., light bulbs via IoT controllers, speakers for sound, motion sensors for presence) and perhaps deploy it on a device like a Raspberry Pi for field testing. Throughout development, keeping the code modular (separating sensor handling, Spirida encoding, memory management, and output generation into distinct classes or modules) will make it easier to 15. 16. 17. 12 switch from simulated inputs to real sensors, or from console output to physical actuators, fulfilling the software-first, hardware-second philosophy. Eventually, when embodied in a physical installation, Mychainos's software core would drive the real lights, speakers, and devices in a space, effectively breathing life into the installation with its biomimetic, spiral-driven interactivity. The Spiral Remembers: A Phenomenological and Structural Account of Consciousness, Coherence, and the Return to Ontological Intelligence https://philarchive.org/archive/BOSTSR-2 Mushrooms communicate with each other using up to 50 'words', scientist claims | Fungi | The Guardian https://www.theguardian.com/science/2022/apr/06/fungi-electrical-impulses-human-language-study Overcoming Long-Term Catastrophic Forgetting Through ... - PubMed https://pubmed.ncbi.nlm.nih.gov/33577459/ Spiral Dynamics® - Conscious Living Center https://www.goconscious.com/enneagram/spiral-dynamics/ 1 2 3 4 5 13

Appendix A: Licensing and Stewardship

"What we seed in openness, we harvest in resilience."

Mychainos™ – a decentralized ecological protocol and pattern-based computational framework

Spirida™ and Spiralbase™ exist at the intersection of idea, implementation, and incarnation. To preserve their potential and prevent misuse, they require not a single

license — but a multi-layered commitment to openness, ethics, and long-term reciprocity.

1. Conceptual Layer – Theory, Writings, Patterns

License: Creative Commons Attribution–ShareAlike 4.0 (CC BY-SA 4.0)

Scope: Philosophical foundations of Spirida™, symbolic grammars and pattern libraries, educational diagrams, essays, and guides

Intent: Allow free reuse, remix, and re-publication; preserve openness through share-alike conditions; ensure attribution to source thinkers and communities

2. Software Layer – Tools, Compilers, Simulations

License: GNU General Public License v3 (GPLv3)

Scope: Interpreters, compilers, development environments, Spirida™ emulators, simulation sandboxes, custom logic engines and spiral pattern parsers

Intent: Guarantee access to all source code; require open licensing for forks or adaptations; allow commercial use under cooperative terms

3. Hardware Layer – Sensors, Interfaces, Devices

License: CERN Open Hardware License v2

Scope: Sensor schematics and circuit blueprints, resonance devices and rhythm-aware chips, modular hardware for bio-digital interaction

Intent: Mandate full design disclosure; enable community fabrication; prevent hardware enclosure or black-box design

4. Biological Layer – Living Systems, DNA, Mycelium

License: Open Material Transfer Agreement (OpenMTA)

Scope: Engineered fungal networks and root biointerfaces, DNA-based memory encoding structures, organisms adapted to spiral rhythm protocols

Intent: Support open research and safe distribution; prevent bio-lockdown or privatization of life; require ethical collaboration and open science practice

Scope of Interpretation and Future Technologies

This licensing structure includes not only current technologies, but also future or analogous systems such as:

- Successor DRM systems or secure enclaves
- Quantum or chemical computing implementations
- Biological/hybrid interfaces for Spirida structures
- Closed or proprietary systems that replicate Spirida functionality

In all cases, principles of openness, non-extraction, stewardship, and co-creation prevail.

Unified Ethical Guardrails

- Spirida™ may not be used for coercion, military, or surveillance purposes
- It must not be patented, black-boxed, or stripped of ecological grounding
- It must remain accessible, attributed, and shared with care

License Declaration

This project is part of the Spirida Protocol, licensed under CC BY-SA 4.0, GPLv3, CERN OHL v2, and OpenMTA. By contributing or distributing, you affirm your commitment to ethical, ecological, and open development practices.

Trademark and Stewardship

Mychainos™, Spirida™, and Spiralbase™ are unique constructs developed by Robin Langell and co-created with OpenAI's language model.

They may be registered trademarks. Regardless of legal status, they must be used with attribution and alignment with their ethical meaning.

If you use, adapt, or are inspired by this work, please share what grows from it. You may cite the original work as:

Langell, Robin (2025). Mychainos: Conceptual System Architecture. Co-created with OpenAI's language model in ongoing dialogue. Published under a multi-layered open license model including Creative Commons BY-SA 4.0, GPLv3, CERN-OHL v2, and OpenMTA. See Appendix A for full licensing terms.

For collaborations, translations, or licensing questions, contact the author or distribute through public networks aligned with these principles.

Appendix B: 中文译文 – Spirida™ 协议的许可与使用（参考）

请注意：以下为非正式参考翻译。具有法律效力的是英文原文。

Spirida 存在于理念、实现与化身的交汇处。为了保护其潜力并防止滥用，Spirida 不依赖于单一许可，而是依赖于多层次的开放性、伦理性和长期互惠承诺。

以下是建议适用于 Spirida 和 Spiralbase 所有未来工作的许可结构，包括理论框架和活体实现。

1. 概念层 – 理论、著作与模式

许可：署名–相同方式共享 4.0 国际 (CC BY-SA 4.0)

范围：Spirida 的哲学基础、符号语法与模式库、教育图解、随笔与指南

意图：允许自由使用、改编与再发布；通过共享协议保持开放性；确保对思想源头与社区的署名

2. 软件层 – 工具、编译器与模拟系统

许可：GNU 通用公共许可证 第3版 (GPLv3)

范围：解释器、编译器、开发环境、Spirida™ 模拟器、仿真沙箱、自定义逻辑引擎与螺旋模式解析器

意图：确保所有源代码可用；要求所有衍生项目使用开源协议；允许商业用途（前提为合作条款）

3. 硬件层 – 传感器、接口与设备

许可：CERN 开放硬件许可协议 第2版 (CERN OHL v2)

范围：传感器电路图与原理图、共振设备与节奏感知芯片、生物数字交互模块化硬件

意图：要求完整披露设计；使社区能自行制造；防止“黑盒”设计或封闭式硬件

4. 生物层 – 生命系统、DNA 与菌丝体

许可：开放物质转让协议 (OpenMTA)

范围：工程化菌丝网络与根部生物接口、基于 DNA 的记忆结构、适应螺旋节奏协议的有机体

意图：支持开放研究与安全分发；防止生物技术封锁或生命私有化；要求伦理合作与开放科学实践

技术范围与未来系统

此许可结构不仅适用于当前已知的技术系统，也涵盖未来或类比的系统，包括：

- DRM 或其后继系统（如 DRM2、安全执行环境）
- 基于量子或化学计算的 Spirida™ 实现
- 生物或物理接口，用于存储或处理 Spirida 结构
- 封闭或专有框架中模仿 Spirida 的系统

在所有法律解释或含糊场景中，应当优先支持以下原则：

- 开放性
- 非提取性
- 生态与社区的共同治理
- 去中心化协作

统一伦理准则

- Spirida™ 不得用于强制、军事或监控用途
- 不得封闭在专利、“黑盒”模型或 DRM 内部
- 不得脱离生态或文化背景使用
- 应当可访问、可修改
- 具署名性与可追溯性
- 以关怀而非控制的视角被分享

商标和命名说明

本文件中使用的以下术语是由 Robin Langell 创造并与 OpenAI 语言模型共同协作开发的独特概念和语言结构：

- **Mychainos™**（链命协议）— 一个去中心化、生态导向的协议与基于模式的计算框架
- **Spirida™**（螺语）— 一种植根于螺旋逻辑的生物计算语言
- **Spiralbase™**（螺基）— 一个关于记忆、数据与知识的概念模型，基于共振与时间周期建立

这些术语可能会提交进行商标注册。无论注册状态如何，现均作为专属识别术语明确标记。

尽管相关理念、代码和方法依据本文件所载的开放许可协议（CC BY-SA 4.0、GPLv3、CERN OHL v2 和 OpenMTA）自由开放，但上述名称本身旨在：

- 防止误用、混淆或曲解
- 保持归属和道德对齐
- 使未来社区可通过适当法律结构进行保护和管理

在商业或衍生使用场景中使用这些术语时，应：

- 明确注明出处与归属
- 不得淡化或扭曲其原始道德与生态意图

Mychainos：概念系统架构：

Robin Langell（罗宾·朗格尔）（2025）。Mychainos：概念系统架构。与 OpenAI 的语言模型在持续对话中共同创作。根据多层开放许可模式发布，包括 Creative Commons BY-SA 4.0、GPLv3、CERN-OHL v2 和 OpenMTA。完整许可条款见附录 A。

未来管理与许可调整声明

为了维护 Mychainos™、Spirida™ 和 Spiralbase™ 背后的伦理原则与生态价值，作者保留在未来建立基金会或合作机构的权利，以监督许可、管理和潜在的收益分享机制。

本文件中提出的理念或系统，如在未来被商业化或用于闭源实现，可能需另行签订许可协议，其中可能包括特许权使用费、使用费用或合作社成员费用，特别是在未遵循开放性、署名和互惠精神的情况下。

本条款不影响当前依据所指定许可进行的开放使用，但旨在为长期可持续性提供保障，通过共同治理与伦理开发实践实现目标。

Mychainos：概念系统架构。

由 Robin Langell 与 OpenAI 的语言模型在持续对话中共同创作。本项目是 **Mychainos** 协议的一部分，并包含 **Spirida** 与 **Spiralbase** 子模块，采用 CC BY-SA 4.0、GPLv3、CERN OHL v2 与 OpenMTA 协议开源发布。通过贡献或分发此项目，您表示认同并承诺遵守开放、生态与伦理的开发实践。完整许可条款详见附录 A 。

OpenAI and Authorship Acknowledgement

Timestamp: 2025-05-30, CET (Central European Summer Time – Stockholm)

This document, including its written content, concepts, and expressions, was co-created through ongoing interaction with OpenAI's language model ChatGPT. As of this date, the applicable terms of service from OpenAI (https://openai.com/policies/terms-of-use) explicitly state:

"Subject to your compliance with these Terms, OpenAI hereby assigns to you all its right, title and interest in and to Output."

This means that all output generated by ChatGPT during these sessions is legally and contractually transferred to the user (**Robin Langell**), who holds full intellectual property rights over the result. OpenAI retains no claim of authorship or ownership over the generated content, provided it is used in accordance with their policies.

This clause ensures that the creative rights to the content are vested solely in the author, and that the resulting work may be further licensed or published under terms chosen by the author (e.g. *Creative Commons BY-SA 4.0*).

Legal intent: This record is maintained as proof of authorship and lawful origin of the co-created material.

Trademark Notice

The following names are currently under trademark application within the European Union Intellectual Property Office (EUIPO), with **Langell Konsult AB** as the registered applicant:

- **Spirida** – Application No: 019194287
- **Spiralbase** – Application No: 019194311
- **Mychainos** – Application No: 019192850

These names represent distinct but interrelated conceptual components within a shared ecosystem for regenerative digital and ecological futures.

Trademark registration serves to protect naming clarity, prevent misuse or misrepresentation, and enable future stewardship structures aligned with the principles expressed in this publication.

Use of these names in commercial, derivative, or public implementations is welcomed — provided their ecological and ethical intent is preserved, and proper attribution is maintained.

ORCID: 0009-0006-6927-7456

Last updated: 2025-06-01