

# Biological Motifs for Agentic Control

A Typed Interface Correspondence between Gene Regulatory Networks and Agentic Software Architectures

Preprint – Feedback Welcome

Bogdan Banu  
bogdan@banu.be

March 30, 2026

## Abstract

The transition of Large Language Models (LLMs) from passive generators to autonomous agents has introduced significant challenges in reliability, security, and state management. Current agentic architectures are often constructed ad-hoc, prone to “hallucination cascades,” infinite loops, and prompt injection attacks. This paper argues that many of these failure modes can be analyzed using control motifs long studied in systems biology, provided the comparison is made at the level of typed interfaces and coordination structure rather than literal biological mechanism.

We develop a typed interface correspondence between Gene Regulatory Networks and agentic software systems using polynomial functors and wiring diagrams. Five biological motifs—Coherent Feed-Forward Loops for noise suppression, Adaptive Immunity for layered security, Mitochondrial Signaling for resource governance, Endosymbiosis for neuro-symbolic integration, and Morphogen Diffusion for spatially varying coordination—are mapped to composable software design patterns. An epistemic topology layer derives Kripke-style knowledge operators from the wiring diagram’s observation structure and proves four predictive theorems for multi-agent scaling.

The core contributions are: (1) the Agentic Operad, a typed syntax for agent composition with provable error suppression bounds for feed-forward topologies; (2) an epistemic topology with four theorems—error amplification, sequential penalty, parallel acceleration, and tool density scaling—whose qualitative predictions are consistent with published multi-agent benchmarks; and (3) a six-layer progression from structure through development, grounded in autonomous learning frameworks and convergence proxies from the empirical literature. A reference implementation with 1,130 tests and 81 examples illustrates practical feasibility.

## 1 Introduction

The field of Artificial Intelligence is undergoing a paradigm shift from Generative AI (systems that produce text based on static prompts) to Agentic AI (systems that execute multi-step workflows to achieve autonomous goals). While the capabilities of individual Large Language Models (LLMs) have scaled predictably, the engineering of systems of agents remains a fragile art. Developers struggle with non-deterministic outputs, infinite loops, adversarial attacks, and the difficulty of maintaining global coherence in distributed, stochastic systems.

We argue that these challenges are not purely ad-hoc engineering problems, but recurring constraints of distributed information processing systems. A useful formal analogue is provided by Gene Regulatory Networks (GRNs). At the atomic level, a gene and an agent capability both expose

typed interfaces that consume local signals and produce downstream effects. At the composite level, a biological cell organizes many genes plus shared infrastructure (organelles, membrane); analogously, an executable agent organizes many capabilities plus shared runtime components. This multi-scale correspondence, rather than a literal identity of mechanisms, is the organizing hypothesis of the paper.

### 1.1 The Biological Heuristic

Biology has evolved specific topological structures, known as Network Motifs, to handle noise, security, and state [27]. We identify five critical biological heuristics that map directly to agentic engineering:

- The Coherent Feed-Forward Loop (CFFL): Acts as a persistence detector to filter out transient noise, analogous to “Human-in-the-Loop” guardrails.
- Quorum Sensing: A distributed consensus mechanism where action is taken only when signal density exceeds a threshold, analogous to Mixture of Experts (MoE) voting.
- Chaperone Proteins: Molecular cages that force proteins to fold correctly, analogous to Schema Validators that enforce structured outputs (JSON).
- Mitochondrial Information Processing: Metabolic constraints acting as a “Motherboard” for decision gating, governing not just energy availability but cognitive control policy.
- Adaptive Immunity: The Self/Non-Self distinction, with MHC-like provenance tagging and Trust-Gated access control, relevant to preventing Prompt Injection attacks and hallucination cascades.

### 1.2 The Categorical Bridge

To move this observation from metaphor to discipline, we utilize Applied Category Theory. We define a category-theoretic model of agentic interfaces using the language of **Poly** (Polynomial Functors) as described by Spivak [36]. We use **Poly** as a common language for typed interfaces and wiring diagrams as a language for composition. The claim is not that GRNs and software agents are identical physical systems, but that selected biological and agentic components can be compared within the same interface-level formalism. A capability is then described not by its weights, but by its interface—a dynamical system consuming observations and producing actions:

$$P_A(y) = \sum_{o \in O} y^{I_o}. \quad (1)$$

### 1.3 Contributions

This paper makes the following contributions:

#### Structural Safety.

1. **A Typed Correspondence:** We establish a disciplined mapping between biological components (Genes, Promoters, Plasmids, Organelles) and software components (Capabilities, Schemas, Tools, Runtimes), including multi-cellular organization for multi-agent systems.

2. **The Agentic Operad:** We define WAgent, a syntax for agent wiring that forbids specific classes of **ill-typed wirings** at the topological level. We prove error suppression bounds for the CFFL topology, explicitly accounting for correlation between error modes.
3. **Wire-Level Optics:** We extend the Lens formalism with Prism (conditional type-based routing) and Traversal (batch element-wise processing) optics, enabling richer signal processing at the wiring level.
4. **Composable Coalgebras:** We make the coalgebraic state machine framework explicit and composable, with parallel/sequential composition and observational equivalence criteria.

### Security and Trust.

5. **Adaptive Immunity:** We formalize the Provenance Functor and Trust-Gated Lens, providing structural resistance to content-level trust forgery, where content-based attacks cannot elevate trust levels.
6. **Pathology & Homeostasis:** We classify agentic failures as biological diseases and derive continuous self-repair mechanisms (Chaperone Loop, Regeneration, Autophagy).
7. **Capability-Gated Tool Acquisition:** The Plasmid Registry implements Horizontal Gene Transfer with capability gating, preventing privilege escalation when agents dynamically acquire tools.

### Epistemic and Metabolic Intelligence.

8. **Epistemic Health:** We define Epilexity (Bayesian Surprise) with operational approximations using embedding similarity and perplexity, connecting agent dynamics to the Free Energy Principle.
9. **Metabolic Intelligence:** We distinguish fast (Apoptosis) and slow (Retrograde Response) interventions, and formalize the Metabolic-Epigenetic Coupling for cost-gated retrieval.
10. **Evolutionary Dynamics:** We situate agentic AI within the Vermeij Trend, identifying three selective pressures (adversarial, complexity, efficiency) that drive architectural evolution.
11. **Morphogen Diffusion:** We formalize spatially varying coordination as a discrete-time dynamical system on the agent graph, producing position-dependent behavior without central control.
12. **Epistemic Topology:** We derive Kripke-style knowledge operators from wiring diagram structure and prove four predictive theorems for multi-agent scaling (error amplification, sequential penalty, parallel acceleration, tool density), then check their qualitative consistency with published architecture-level results.

### Optimization.

13. **Diagram Optimization:** Cost-annotated wiring diagrams admit categorical rewriting rules that preserve observational equivalence while improving resource utilization.

By viewing agentic engineering through the lens of theoretical biology and category theory, we aim to provide a framework for building robust software systems whose stability properties derive in part from their network topology.

## 2 Related Work

This work sits at the intersection of Systems Biology, Applied Category Theory, and Agentic AI. While significant research exists within each domain, the formal synthesis of biological control topologies with agentic software architectures has received limited attention.

### 2.1 Network Motifs in Systems Biology

The concept of “Network Motifs”—statistically over-represented sub-graphs in complex networks—was introduced by Milo et al. [27]. Their work demonstrated that biological networks are not random but are composed of specific building blocks selected for functional data processing. Alon [5] further characterized the dynamical properties of these motifs, identifying the Coherent Feed-Forward Loop (CFFL) as a persistence detector. We extend this by mapping these motifs to the stochastic nature of Generative AI.

### 2.2 Applied Category Theory (ACT)

To formalize network structure, we draw upon ACT. Fong and Spivak [16] provide a comprehensive introduction to Applied Category Theory. Spivak [36] and Vagner et al. [37] established a rigorous framework for modeling Open Dynamical Systems using the category **Poly** and the Operad of Wiring Diagrams. Niu and Spivak [29] develop the full mathematical theory of polynomial functors as a language for interaction. To our knowledge, this is the first application of Polynomial Functors specifically designed to model the interface of LLM Agents and to verify safety properties in Agentic topologies.

### 2.3 Reliability in Agentic AI

Techniques such as “Chain of Thought” [40] utilize iterative looping to improve output quality. However, these methods operate primarily at the level of the prompt (the input signal) rather than the topology (the wiring). By importing the concept of Autopoiesis [26], we propose a methodology where reliability is a property of the network architecture itself.

### 2.4 Epistemic Logic in Distributed Systems

Fagin et al. [14] established the foundational framework for reasoning about knowledge in multi-agent systems, formalizing what agents know and what they know about each other’s knowledge. Halpern and Moses [18] proved the impossibility of attaining common knowledge in asynchronous systems, a result with deep implications for coordination protocols. More recently, Davis [11] integrated epistemic and temporal logic within TLA+ for distributed system verification, demonstrating that formal epistemic reasoning can yield practical verification tools. To our knowledge, these formal methods from epistemic logic have not previously been applied to the design and optimization of multi-agent AI architectures.

### 2.5 Temporal Databases and Bi-Temporal Data Models

Bi-temporal data management—tracking both *valid time* (when a fact is true in the world) and *transaction time* (when the system recorded it)—has been studied extensively in the database community. Snodgrass [35] provided the foundational treatment, distinguishing valid-time, transaction-time, and bi-temporal relations, and demonstrating that the two time axes are orthogonal: a fact

may be recorded before it becomes valid, corrected after it ceases to be valid, or both. SQL:2011 [23] standardized temporal query support, including `FOR SYSTEM_TIME` and `FOR BUSINESS_TIME` clauses.

## 2.6 Adaptive Multi-Agent Assembly and Meta-Control

Dupoux, LeCun, and Malik [13] propose a three-system cognitive architecture: System A (observational, statistical, cheap) discovers representations, System B (action-oriented, goal-directed, expensive) discovers causal structure, and System M (meta-control) routes data between them. This tripartite structure maps naturally onto Operon’s fast/deep nucleus distinction and motivates the **WatcherComponent** as a concrete instantiation of System M (§6.3).

Hao et al. [19] demonstrate empirically that incorrect multi-agent runs require systematically more routing decisions than successful ones (e.g., 9.4 vs 7.3 mean decisions on planning tasks). This finding grounds our use of intervention count as a convergence proxy: when the watcher’s cumulative retry/escalate count exceeds a threshold relative to stage count, it emits a non-convergence signal and halts the organism (§10.8).

Jiang et al. [20] provide a structure-oriented taxonomy of Memory-Augmented Generation (MAG) systems, categorizing them into lightweight semantic, entity-centric, episodic/reflective, and structured/hierarchical designs. Their empirical analysis reveals that append-only memory architectures are significantly more robust to backbone format errors than complex structured alternatives—a finding that validates Operon’s design decision to make **BiTemporalMemory** append-only. They also quantify the “Agency Tax” (latency and token overhead of memory maintenance), highlighting a practical concern that the convergence with operational runtimes must address.

Lin et al. [24] propose MemMA, a multi-agent framework coordinating the full memory cycle through a planner-worker architecture with in-situ self-evolution. Their Meta-Thinker provides strategic guidance for both memory construction and retrieval, analogous to Operon’s **WatcherComponent** providing intervention decisions. Their probe-based memory verification—generating synthetic questions after each session to test memory fidelity, then repairing failures immediately—complements Operon’s `counterfactual_replay()`, which detects corrections but does not actively probe for gaps.

Feng, Wang, and Zhu [15] propose Self-evolving Embodied AI, a paradigm comprising five co-evolving components: memory self-updating, task self-switching, environment self-prediction, embodiment self-adaptation, and model self-evolution. These map directly onto Operon’s phased roadmap (bi-temporal memory, adaptive assembly, sleep consolidation, developmental staging, and social learning respectively), and their emphasis on multi-timescale closed-loop co-evolution aligns with Operon’s per-stage (watcher), per-run (adaptive), and per-batch (consolidation) adaptation cycles.

We apply bi-temporal data management [35, 23] to agentic memory in §3.5 and §7.1.

## 3 The Mapping: Biology $\leftrightarrow$ Software

To compare Agentic Systems and Gene Regulatory Networks (GRNs) under a common typed-interface abstraction, we map selected components from both domains to a shared mathematical object. We utilize the category **Poly**, where objects are polynomial functors representing interfaces, and morphisms represent interaction protocols. The claim in this section is a correspondence of interface descriptions, not a proof that the full biological and software domains are equivalent in mechanism, implementation, or dynamics.

### 3.1 Preliminaries: The Category Poly

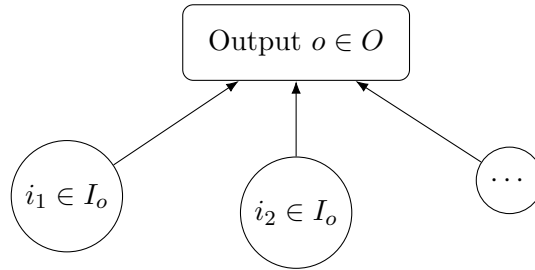
In Applied Category Theory, a Polynomial Functor  $P$  represents a typed interface for a dynamical system. It is defined as a sum of representable functors:

$$P(y) = \sum_{o \in O} y^{I_o}. \quad (2)$$

Here,  $O$  is the set of possible Positions (or Outputs) the system can expose. For each position  $o \in O$ , there is a set  $I_o$  of Directions (or Inputs) required to transition the system to a new state.

- The coefficient  $o$  represents the value produced by the system.
- The exponent  $I_o$  represents the capacity to receive information from the environment.

This formalism captures the essence of a “stateful interface”: the system outputs a value  $o$  and then waits for a specific type of input  $i \in I_o$  before it can proceed.



The Interface  $P(y)$

Figure 1: A visual representation of a Polynomial Functor (often called a “Mushroom” or “Corolla”). The system offers an Output (the cap) and exposes specific Input ports (the stalks) dependent on that output.

### 3.2 The Correspondence: Genes and Agent Capabilities

We now apply this abstract definition to our specific domains.

**Definition 1** (The Gene Object). *A gene  $G$  is a polynomial functor where  $O_G$  is the set of expressed proteins and  $I_G = (I_{prot})_{prot \in O_G}$  is the **family** of regulatory-signal sets (transcription factors) available at each expressed protein:*

$$P_{Gene}(y) = \sum_{prot \in O_G} y^{I_{prot}}. \quad (3)$$

**Definition 2** (The Agent-Capability Object). *An agent capability  $A$  is a polynomial functor where  $O_A$  is the set of generated messages/actions, and  $I_A = (I_{action})_{action \in O_A}$  is the **family** of observation sets available at each action:*

$$P_{Agent}(y) = \sum_{action \in O_A} y^{I_{action}}. \quad (4)$$

**Multi-Scale Composition.** The polynomial functor formalism applies at every level of biological and software organization. Definitions 1 and 2 establish the *atomic* correspondence: a single gene and a single agent capability share the same interface form. Polynomial functors then compose—via the parallel ( $\otimes$ ), serial ( $\circ$ ), and trace ( $\text{Tr}$ ) operations of §4—allowing the same interface language to describe progressively richer assemblies:

Biological Scale	Software Scale	Composition
Gene	Agent capability	Atomic polynomial functor
Cell	Agent runtime	Composite of capabilities + organelles
Tissue	Multi-agent subsystem	Wiring diagram of agents (§6)

A cell is not merely a bag of genes; it is a structured composition with shared infrastructure (organelles) and a boundary (membrane). Likewise, an agent runtime is a structured composition of capabilities with shared components (LLM provider, memory, error handling) and a security boundary. To reduce ambiguity, the rest of the paper uses *capability* for the atomic interface and *agent* for the composed runtime-level object, even though later wiring-diagram examples sometimes use “agent” informally for executable boxes. The organelle mappings below describe this cell-level architecture. The multi-cellular organization of §6 then composes agents into tissues via wiring diagrams—the same formalism, one level up.

### 3.3 The Interface: Promoters as Lenses

In biology, a gene is not universally accessible. It is guarded by a Promoter Region—a specific DNA sequence that only binds to compatible Transcription Factors. In software, an agent is guarded by an API Schema or Context Window definition.

We model this gating mechanism using Optics, specifically Lenses. A Lens consists of two maps between a global state  $S$  and a local view  $V$ :

1. Get (View):  $\text{get} : S \rightarrow V$  (Extracting relevant signal from global state).
2. Put (Update):  $\text{put} : S \times V \rightarrow S$  (Updating global state based on local change).

The “Promoter” acts as a filter that determines which part of the global cellular environment ( $S$ ) is visible ( $V$ ) to the gene.

- **Biological Lens:** The promoter filters the chaotic cellular soup, allowing the gene to “see” only specific molecules (e.g., Lac Repressor).
- **Agentic Lens:** The Context Window filters the massive vector database, allowing the agent to “see” only the relevant retrieved chunks (RAG).

If the input signal does not match the Schema (Promoter), the Lens fails to focus, and the interaction is routed to an explicit **inactive/error** case (equivalently, one works with a **partial** lens, or a total lens into  $V + \text{Error}$ ) (the agent does not run; the gene is not expressed).

### 3.4 Wire-Level Optics: Beyond Lenses

The Lens formalism models constitutive access: a promoter that either admits or blocks a signal. Biological systems employ richer signal-processing at the interface level. We extend the wiring diagram with two additional optic types from the categorical optics literature.

**Prism: Receptor Specificity.** A membrane receptor does not merely pass or block signals; it selects signals by molecular shape. A Prism on a wire admits values of specific data types and rejects others:

$$\text{prism}_A(\tau, v) = \begin{cases} v & \text{if } \tau \in A \\ \perp & \text{if } \tau \notin A \end{cases} \quad (5)$$

where  $A \subseteq T$  is the set of accepted types. This enables fan-out routing: a single output port connects to multiple wires, each guarded by a different prism, directing JSON to one handler and errors to another—analogous to how different receptor types on a cell surface route different ligands to different intracellular pathways.

**Traversal: Polymerase Processivity.** A ribosome does not translate an entire mRNA at once; it processes codons sequentially, applying the same read-translate operation to each element. A Traversal maps a transform  $f$  over collection elements on a wire:

$$\text{traversal}_f(\mathbf{x}) = [f(x_i) \mid x_i \in \mathbf{x}] \quad (6)$$

This models batch processing at the wire level: a list of candidate outputs is transformed element-wise before reaching the downstream agent.

**Composition and Coexistence.** Optics compose: a **ComposedOptic** applies its constituent optics left-to-right, transmitting only if all accept. A wire may carry both a DenatureFilter (§5.3) and an Optic, applied in sequence: denaturation strips syntactic structure, then the optic routes or transforms the sanitized content. This layered processing models the biological reality that signal reception involves multiple sequential steps (ligand binding  $\rightarrow$  receptor conformational change  $\rightarrow$  intracellular cascade).

### 3.5 Epigenetics and State: The Coalgebra

Neither genes nor agents are stateless functions. They possess memory.

- **Biology:** Epigenetic markers (Methylation, Histone modification) alter how a gene responds to signals without changing the DNA code.
- **Software:** Retrieval Augmented Generation (RAG) and Conversation History alter how an agent responds to a prompt without changing the LLM weights.

We model this as a Coalgebra for the polynomial functor  $P$ . A dynamical system is defined as a tuple  $(S, \phi)$ , where  $S$  is the state space and  $\phi$  is the structure map:

$$\phi : S \rightarrow P(S). \quad (7)$$

By expanding  $P(S)$ , we derive the two fundamental operations of the state machine:

1. Readout:  $S \rightarrow O$  (Given current state/memory, what action do I take?)
2. Update:  $\sum_{s \in S} I_{o(s)} \rightarrow S$  (Given current state  $s$  and a new input  $i \in I_{o(s)}$  compatible with its current output  $o(s)$ , what is my new state?)

By establishing this formal dictionary (Table 1), we can compare selected GRN components and agentic components as instances of the same abstract class of typed dynamical interfaces **under this interface-level abstraction**.



**Composable Coalgebras.** The coalgebra formalism becomes most useful when made composable. We define two composition operations that mirror the parallel and serial composition of the Operad (§4):

- **Parallel Coalgebra:** Given  $(S_1, \alpha_1)$  and  $(S_2, \alpha_2)$ , their parallel composition has state  $S_1 \times S_2$  and applies both readout/update operations to the same input—like two signaling pathways activated by the same ligand.
- **Sequential Coalgebra:** Given  $(S_1, \alpha_1)$  over  $P_1$  and  $(S_2, \alpha_2)$  over  $P_2$ , the sequential composition pipes the readout of the first as input to the second, with joint state  $S_1 \times S_2$ —modeling signal transduction cascades.

**Bisimulation: Observational Equivalence.** For the deterministic state machines considered here, we use the following observational criterion: two state machines  $(S_1, \alpha_1)$  and  $(S_2, \alpha_2)$  are equivalent if, for every input sequence, they produce identical output sequences. We write  $M_1 \sim M_2$  when

$$\forall \mathbf{i} \in I^* : \text{readout}_1^*(\mathbf{i}) = \text{readout}_2^*(\mathbf{i}) \quad (8)$$

where  $\text{readout}^*$  denotes the lifted readout over the input sequence. A diverging input (witness) constitutes a proof of non-equivalence. This supports formal comparison of deterministic implementations within the input model considered here; stronger coinductive verification claims are left for future work.

**Temporal Coalgebra: Bi-Temporal State.** The coalgebra above models state at a single point in time. In practice, agents accumulate beliefs that may be corrected retroactively. We extend the state space to carry two independent time indices. Let  $\mathcal{T} = (\mathbb{R}_{\geq 0}, \leq)$  denote a totally ordered time domain. A *bi-temporal state* augments the coalgebra with a pair of interval-valued annotations:

$$S_{\text{bt}} = S \times \underbrace{[\mathcal{T}, \mathcal{T} \cup \{\infty\})}_{\text{valid interval}} \times \underbrace{[\mathcal{T}, \mathcal{T} \cup \{\infty\})}_{\text{record interval}} \quad (9)$$

where an open-ended interval  $[t, \infty)$  denotes a currently active fact. The key invariant is *append-only correction*: closing a record’s transaction interval and appending a new record with a **supersedes** pointer, rather than mutating the original. This ensures that for any pair  $(t_v, t_r)$ , the *belief state*—the set of facts valid at  $t_v$  and known at  $t_r$ —is uniquely reconstructible by filtering on both intervals simultaneously.

The readout function becomes time-parameterized:  $\text{readout}(s, t_v, t_r)$  returns only those facts whose valid interval contains  $t_v$  and whose record interval contains  $t_r$ . This separates two questions that a single-time coalgebra conflates: “what is true now?” (valid-time query) versus “what did the system believe at decision time?” (bi-temporal query). The implementation (§10.8) instantiates this as `BiTemporalMemory` with explicit `retrieve_valid_at`, `retrieve_known_at`, and `retrieve_belief_state` methods.

### 3.6 Metabolic Coalgebras: Formalizing Resource Constraints

Finally, we address the physical constraints of computation. Just as biological systems are limited by ATP availability [25, 21], agentic systems are limited by token budgets and latency. To model this, we extend our coalgebraic framework to include resource constraints, defining a **Metabolic Coalgebra**. Mathematically, this is an instance of a Quantitative Coalgebra enriched over a

Gene (Function)	Agent (LLM + Tools)
Transcription Factors ( $I$ )	Observations ( $I$ )
Proteins ( $O$ )	Actions ( $O$ )
Promoter Binding	Schema Match
Expression	Generation

Figure 2: The Structural Correspondence. Both Genes and Agent Capabilities act as transducers converting Input Contexts ( $I$ ) into Output Expressions ( $O$ ), governed by the same categorical laws (at the level of polynomial-interface models).

Category Concept	Biological Realization (GRN)	Software Realization (Agentic)
Polynomial Functor ( $P$ )	Gene Interface	Agent Interface (System Prompt)
Output Position ( $O$ )	Protein Expression	Tool Call / Message
Input Direction ( $I$ )	Transcription Factor Binding	Observation / User Prompt
Lens (Optic)	Promoter Region	API Schema / Context Window
Internal State ( $S$ )	Epigenetic Markers (Methylation)	Vector Store / Chat History
Morphism ( $\circ$ )	Signal Transduction Pathway	Data Pipeline
<i>Organelles (Specialized Processing Units)</i>		
Template Engine	Ribosome (mRNA $\rightarrow$ Protein)	Prompt Template Factory
Output Validation	Chaperone (Protein Folding)	Schema Validator / JSON Parser
Waste Processing	Lysosome (Autophagy)	Error Handler / Garbage Collector
Decision Center	Nucleus (Transcription)	LLM Provider Wrapper
Input Filter	Membrane (Immune System)	Prompt Injection Defense
Computation Engine	Mitochondria (MIPS)	Runtime Supervisor / Decision Gate
<i>Lifecycle and Rhythms</i>		
Lifespan Limit	Telomere Shortening	Operation Counter / Max Iterations
Periodic Scheduling	Circadian Oscillator	Health Checks / Heartbeat

Table 1: The Correspondence Dictionary (Extended)

resource monoid, effectively restricting the domain of the state transition function to resource-sufficient states.

We align this definition with the theory of **Quantitative Polynomial Functors** [28], treating the system as a state machine enriched over a resource monoid.

**Definition 3** (The Resource Monoid). *Let  $(\mathcal{R}, +, 0, \geq)$  be an ordered commutative monoid representing computational resources, equipped with a partial subtraction  $r \ominus c$  defined whenever  $r \geq c$ . For a single resource dimension (e.g., token counts),  $\mathcal{R} \cong \mathbb{N}$ ; for multi-dimensional accounting (latency, memory),  $\mathcal{R} \cong \mathbb{R}_{\geq 0}^k$ .*

**Definition 4** (Metabolic Coalgebra). *A resource-constrained agent is a coalgebra  $(S, \alpha)$  over a polynomial functor  $P$ , where the state space is the product of the logical state  $L$  and the resource state  $\mathcal{R}$ :*

$$S \cong L \times \mathcal{R}$$

*The structure map  $\alpha : S \rightarrow P(S) + \perp$  is defined as a **partial map** guarded by cost. Writing only the continuation state and suppressing the output/readout coordinate of  $P(S)$ , a transition requiring*

Biological Concept	Agentic Concept	Formal Structure
<i>Immunity and Security</i>		
Toll-Like Receptor (TLR)	Regex Injection Filter	Pattern Matching
MHC Presentation	Provenance Labeling	Functor $\mathcal{P} : \mathbf{Msg} \rightarrow \mathbf{Trust}$
T-Cell Receptor	Trust Gate	Partial Lens
Negative Selection	Injection Training	Penalized Learning
Regulatory T-Cell	Confidence Dampening	Suppression Function
Immune Memory	Threat Signature Store	Hash-Indexed Cache
<i>Metabolism and Control</i>		
ATP	Token Budget	Resource Monoid $\mathcal{R}$
mPTP Opening	Fast Apoptosis Trigger	Guard Condition
Retrograde Signaling	Phenotype Reshaping	Slow Adaptation
Metabolic-Epigenetic State Coupling	Cost-Gated Retrieval Policy	Conditional Access Control
<i>Information and Health</i>		
Trophic Factors	Novel Input Signals	Epiplexity $> \delta$
Bayesian Brain	Free Energy Minimization	KL Divergence
Apoptosis	Agent Termination	State $\rightarrow \perp$
<i>Multi-Cellular Organization</i>		
Genome	Base Model Weights	Shared Parameters
Epigenome	System Prompt + RAG	Phenotype Context
Morphogen Gradient	Shared Context Variables	JSON State
Morphogen Diffusion	Gradient Propagation on Agent Graph	Discrete PDE on $G$
Tissue Boundary	Trust Boundary / Capability Ceiling	Type Barrier
Prism Optic	Conditional Wire Routing	Partial Optic
Traversal Optic	Batch Wire Transform	Endofunctor on Lists
Bisimulation	Agent Equivalence Testing	Observational Equivalence

Table 2: Extended Correspondence Dictionary: Security, Metabolism, and Organization

cost  $c \in \mathcal{R}$  has the guarded form:

$$\alpha_{\text{cont}}(l, r) = \begin{cases} (l', r \ominus c) & \text{if } r \geq c \\ \perp & \text{if } r < c \quad (\text{Apoptosis}) \end{cases}$$

This structure maps to the energetics of transcriptional elongation. A gene or agent capability cannot produce its output instantaneously; it must carry out a sequence of state transitions, each with a cost. The Metabolic Coalgebra models this dependency: if the cellular or computational energy budget is exhausted, execution stalls (Ischemia), and the component fails to execute its function, regardless of its regulatory logic.

This formalism establishes that “Ischemia” (Token Starvation) is not merely a runtime error, but a reachable terminal state  $\perp$  in the system’s dynamics. This mirrors the biological mechanism where failure to meet metabolic costs triggers p53-mediated apoptosis [7].

**Theorem 1** (The Metabolic Bound (Qualified)). *Assume either a single scalar resource budget  $\mathcal{R} \cong \mathbb{N}$  or, more generally, a well-founded scalar potential  $\mu : \mathcal{R} \rightarrow \mathbb{R}_{\geq 0}$  such that every non-identity transition of cost  $c$  satisfies  $\mu(r \ominus c) \leq \mu(r) - c_{\min}$  for some  $c_{\min} > 0$ . Assume also that the resource*

state is monotone nonincreasing (no regeneration), and that infinite executions may not consist solely of stutter / identity steps. Then any execution contains at most  $\lfloor \mu(R_{\text{total}})/c_{\min} \rfloor$  non-identity transitions; in the scalar-budget case this is  $\lfloor R_{\text{total}}/c_{\min} \rfloor$ . Hence termination is decidable up to stuttering. If regeneration, zero-cost cycles, or unrestricted stutter loops are permitted, termination requires an additional well-founded potential or an explicit budget/termination certificate [8].

**Operational takeaway.** For an agent-systems reader, the theorem says: if every non-trivial step burns at least a fixed minimum amount of budget and budget never regenerates, then execution length is bounded by initial budget divided by that minimum burn rate. Infinite behavior becomes possible only through free stutter, zero-cost cycles, or regeneration.

*Proof sketch.* Let the initial resource state be  $R_{\text{total}}$ . By assumption, every non-identity transition decreases the scalar potential  $\mu$  by at least  $c_{\min} > 0$ . After  $N$  non-identity transitions we therefore have

$$\mu(r_N) \leq \mu(R_{\text{total}}) - Nc_{\min}.$$

Because  $\mu$  is nonnegative, the right-hand side must remain nonnegative, which implies

$$N \leq \left\lfloor \frac{\mu(R_{\text{total}})}{c_{\min}} \right\rfloor.$$

In the scalar-budget case this is exactly  $\lfloor R_{\text{total}}/c_{\min} \rfloor$ . Hence only finitely many non-identity transitions are possible. Under the stated no-regeneration / no-infinite-stutter assumptions, this yields decidability of termination up to stuttering. If those assumptions are removed, the argument no longer forces progress, so an additional well-founded ranking or explicit budget certificate is required.  $\square$

### 3.7 Additional Organelles: Completing the Cellular Architecture

Beyond the core interface mapping, biological cells contain specialized organelles that handle distinct aspects of cellular function. We extend the correspondence to four additional structures that map directly to agentic software components.

**Ribosome: Template-to-Output Synthesis.** In biology, the ribosome reads messenger RNA (mRNA) sequences and synthesizes proteins by assembling amino acids according to the genetic code. Transfer RNA (tRNA) molecules carry amino acids to the ribosome, where codons (three-nucleotide sequences) specify which amino acid to add.

In agentic systems, the Ribosome maps to a **prompt template engine**:

- **mRNA**  $\rightarrow$  Prompt templates with variable slots
- **tRNA**  $\rightarrow$  Context bindings (variable  $\rightarrow$  value mappings)
- **Codons**  $\rightarrow$  Template directives (variables, conditionals, loops)
- **Translation**  $\rightarrow$  Template rendering with context injection

Just as the ribosome ensures that the genetic code is faithfully translated into functional proteins, the software ribosome ensures that abstract prompt templates are instantiated into concrete, well-formed prompts.

**Lysosome: Waste Processing and Recycling.** The lysosome is the cell’s recycling center, containing enzymes that break down cellular waste, damaged organelles, and foreign material. Through autophagy, the cell digests its own components to recover building blocks during stress.

In agentic systems, the Lysosome maps to **error handling and garbage collection**:

- **Waste Classification** → Categorizing failures (timeout, validation error, toxic input)
- **Digestion** → Processing errors to extract debugging information
- **Recycling** → Recovering useful context from failed operations
- **Autophagy** → Periodic cleanup of stale cache and expired state
- **Toxic Disposal** → Secure handling of sensitive data (API keys, PII)

The lysosome prevents accumulation of “cellular debris” that could poison the system—analogous to memory leaks or error cascades in software.

**Nucleus: The Decision Center.** In eukaryotic cells, the nucleus houses the DNA and serves as the control center for gene expression. Transcription factors enter the nucleus, bind to promoter regions, and initiate transcription of specific genes.

In agentic systems, the Nucleus maps to the **LLM provider wrapper**:

- **DNA** → Pre-trained model weights (static substrate of capability)
- **Transcription** → Inference (prompt → response generation)
- **Nuclear Envelope** → Provider abstraction layer (API boundary)
- **Nucleolus** → Tool integration hub (where external capabilities are assembled)

The nucleus abstracts the complexity of the underlying LLM, exposing a consistent interface regardless of the provider (Anthropic, OpenAI, Gemini).

**Telomere: Lifecycle and Senescence.** Telomeres are protective caps at the ends of chromosomes that shorten with each cell division. When telomeres become critically short, the cell enters senescence (permanent growth arrest) or apoptosis. The enzyme telomerase can extend telomeres, enabling continued division in stem cells.

In agentic systems, Telomeres map to **lifecycle management**:

- **Telomere Length** → Remaining operation budget (max iterations)
- **Shortening** → Decrementing counter per operation
- **Senescence** → Graceful degradation (reduced capability mode)
- **Apoptosis** → Clean shutdown when budget exhausted
- **Telomerase** → Renewal mechanism (resetting counters for trusted agents)

This provides a biological basis for the common pattern of limiting agent iterations. Rather than arbitrary timeouts, the telomere model frames lifecycle limits as a natural property of the system’s “cellular age.”

**Mitochondria: The Metabolic Motherboard.** Recent scholarship reframes mitochondria not merely as the cell’s powerhouse, but as the **Mitochondrial Information Processing System (MIPS)** [33]. Mitochondria sense environmental stress and integrate metabolic signals to govern cellular decision-making. They function as “social signaling organelles” that communicate with the nucleus and other organelles.

In agentic systems, the Mitochondria maps to the **Runtime Supervisor**:

- **ATP Production** → Deterministic computation (tool execution, code evaluation)
- **Stress Sensing** → Monitoring token burn rate vs. informational yield
- **Retrograde Signaling** → Forcing strategy shifts when metabolic efficiency drops
- **Fusion/Fission** → Context fusion under resource constraints

**Temporal Dynamics: Fast vs. Slow Interventions.** A crucial distinction exists between the timescales of mitochondrial intervention. In biology, retrograde signaling influences nuclear gene expression over hours to days—it is a *developmental* response that reshapes the cell’s phenotype. In contrast, acute metabolic stress (ATP depletion,  $\text{Ca}^{2+}$  overload) triggers immediate responses: cytochrome c release initiates apoptosis within minutes.

We preserve this distinction in the agentic mapping:

- **Fast Intervention (Acute):** The Runtime monitors real-time metrics (token velocity, error rate). When thresholds are breached, it triggers immediate Apoptosis—terminating the current chain-of-thought without negotiation. This mirrors the mitochondrial permeability transition pore (mPTP) opening.
- **Slow Intervention (Chronic):** The Runtime accumulates statistics across sessions (average efficiency, failure patterns). These inform *Retrograde Responses*—modifications to system prompts, retrieval strategies, or model selection that reshape the agent’s “phenotype” over deployment cycles. This mirrors how chronic metabolic stress induces mitochondrial biogenesis and metabolic reprogramming.

The Runtime does not “override” the LLM in the sense of injecting tokens mid-generation; rather, it governs the *boundary conditions* within which generation occurs, and triggers state transitions (continue, pivot, terminate) at defined checkpoints.

## 4 Formal Syntax: The Agentic Operad

To formalize admissible agent compositions, we define a typed operad of wiring diagrams, denoted as **WAgent**. Here the operad serves primarily as a syntax: a “grammar” for connecting operations (boxes) via typed wires. It specifies which agent topologies are well-formed. When we make probabilistic or behavioral claims below, those claims rely on additional assumptions about the implementations inhabiting the boxes, not on the operadic syntax alone [37].

### 4.1 The Typing Rules

In **WAgent**, every wire carries a specific Type  $\tau \in T$ .

$$T = \{\text{Text, JSON, Image, Error, ToolCall, Stop, Approval}\}. \quad (10)$$

These types correspond to biological molecular specificity (e.g., a specific transcription factor only binds to a specific DNA sequence). A connection is valid if and only if the type of the output port of Agent  $A$  matches the type of the input port of Agent  $B$ .

## 4.2 The Composition Operations

The operad defines three fundamental operations for combining agents. Any complex agentic architecture, no matter how large, can be decomposed into these three primitives.

### 4.2.1 Parallel Composition ( $\otimes$ )

Two agents,  $A$  and  $B$ , execute simultaneously with no information exchange:

$$A \otimes B. \quad (11)$$

- **Biological Analogy:** Two genes located on different chromosomes expressing proteins independently.
- **Constraint:** This operation is valid only if the internal state spaces  $S_A$  and  $S_B$  are disjoint. If they share a mutable memory store, the operation leaves the **independent-state interpretation** and requires an explicit **resource-sharing** structure (e.g., a shared state component or a Resource Sharing decorator).

### 4.2.2 Serial Composition ( $\circ$ )

The output of Agent  $A$  is piped directly into the input of Agent  $B$ :

$$B \circ A. \quad (12)$$

- **Biological Analogy:** A Signal Transduction Pathway (Protein A activates Protein B).
- **Static Type Checking:** This allows for static type checking of agent graphs. If Agent  $A$  outputs Natural Language but Agent  $B$  expects JSON Schema, the composition is undefined in WAgent. This moves runtime **type/schema mismatch** errors to “compile-time” architectural errors.

### 4.2.3 Contraction / Trace ( $Tr$ )

A feedback loop where an output port of Agent  $A$  is wired back into one of its own input ports:

$$Tr(A). \quad (13)$$

- **Biological Analogy:** Autoregulation (Homeostasis) or Positive Feedback.
- **Software Implication:** Trace captures **explicit feedback wiring** (outputs fed back as inputs), e.g., when an agent conditions on its own prior outputs or a memory buffer. Internal chain-of-thought is modeled as hidden state in the coalgebra, not by Trace itself.

## 4.3 Theorem: Topological Error Suppression

We now combine the wiring syntax with a simple stochastic failure model to analyze when the Coherent Feed-Forward Loop (CFFL) suppresses errors more effectively than a direct serial connection on high-stakes tasks.

**Reading convention.** For readers coming from AI agent systems rather than category theory, each result in this section should be read in three layers: the wiring diagram tells us which signals or approvals can reach an action sink; the stochastic or trust model assigns probabilities or integrity levels to those signals; the proof then converts that structural constraint into a failure or authorization bound.

**Network Motif 1 (Coherent Feed-Forward Loop).** A topological structure where Signal  $X$  activates  $Z$  directly, but also activates  $Y$  which gates  $Z$ . The node  $Z$  functions as an AND gate: it fires if and only if  $X \wedge Y$ .

**Theorem 2** (Error Suppression in CFFL). *Let  $A_{\text{gen}}$  be a generator agent and  $A_{\text{ver}}$  be a verifier agent. Let  $E_{\text{gen}}$  denote the event that the generator emits an erroneous candidate, and let  $M_{\text{ver}}$  denote the event that the verifier misses that error and still emits an approval token.*

- **Case 1: Direct Link (Serial).** *The system fails if  $A_{\text{gen}}$  hallucinates.*

$$P(\text{Fail}_{\text{direct}}) = P(E_{\text{gen}}).$$

- **Case 2: CFFL Topology (Independent).** *Under the assumption of independence:*

$$P(\text{Fail}_{\text{CFFL}}) = P(E_{\text{gen}}) \times P(M_{\text{ver}}).$$

- **Case 3: CFFL Topology (Correlated).** *In practice, generator errors and verifier misses are often correlated—if the generator hallucinates a plausible-sounding function, the verifier (especially if using the same base model or training distribution) may be more likely to accept it. Let  $\rho$  be the phi coefficient between the Bernoulli variables  $E_{\text{gen}}$  and  $M_{\text{ver}}$ , with  $p = P(E_{\text{gen}})$  and  $q = P(M_{\text{ver}})$  (assume  $p, q \in (0, 1)$  so  $\rho$  is well-defined). Then:*

$$P(E_{\text{gen}} \wedge M_{\text{ver}}) = pq + \rho \sqrt{p(1-p)q(1-q)}.$$

*Because  $E_{\text{gen}}$  and  $M_{\text{ver}}$  are Bernoulli,  $\rho$  is **not free in**  $[-1, 1]$ ; it is constrained by the Fréchet–Hoeffding bounds*

$$P(E_{\text{gen}} \wedge M_{\text{ver}}) \in [\max(0, p + q - 1), \min(p, q)],$$

*equivalently*

$$\rho \in \left[ \frac{\max(0, p + q - 1) - pq}{\sqrt{p(1-p)q(1-q)}}, \frac{\min(p, q) - pq}{\sqrt{p(1-p)q(1-q)}} \right].$$

**Assumptions.** The theorem uses the minimal failure model appropriate for a two-stage execution gate: in the direct serial case, an erroneous action is emitted exactly when the generator emits an erroneous candidate; in the CFFL case, an erroneous action is emitted exactly when the generator emits an error *and* the verifier still approves it.

**Corollary 1** (Correlation Degradation). *Let  $p = P(E_{\text{gen}}) = P(M_{\text{ver}})$  for simplicity. The failure probability becomes:*

$$P(\text{Fail}) = p^2 + \rho p(1-p) = p^2(1-\rho) + \rho p$$

*with  $\rho \in \left[ -\min\left(\frac{p}{1-p}, \frac{1-p}{p}\right), 1 \right]$ . When  $\rho = 0$  (independence), we recover  $p^2$ . As  $\rho$  increases toward its upper bound, the gain vanishes and  $P(\text{Fail}) \rightarrow p$ .*



**Architectural Implications.** This result makes precise when the CFFL topology provides genuine safety benefits:

- **High correlation:** Reusing the same model family, prompt structure, or evidence sources can make verifier misses positively correlated with generator errors, erasing much of the gain.
- **Heterogeneous generators and verifiers:** Prompt diversity or model-family diversity can reduce correlation, but the magnitude is empirical rather than universal.
- **Orthogonal checks:** Symbolic or tool-grounded verifiers are a useful limiting case because their failure modes need not mirror the generator’s linguistic ones.

These are architectural heuristics rather than fitted universal ranges for  $\rho$ . The topological structure (conjunctive gating) is necessary but not sufficient; diversity in the components populating that topology determines the actual error suppression achieved.

**Operational takeaway.** For agent designers, the result is simple: adding a reviewer buys multiplicative safety only when the reviewer is not making the same mistakes as the generator. The AND gate gives the multiplicative structure; correlation gives that improvement back.

*Proof sketch.* In the direct serial topology there is only one path from the generator to the action sink, so an erroneous final action occurs iff the generator emits an erroneous candidate. Thus

$$\text{Fail}_{\text{direct}} = E_{\text{gen}}, \quad P(\text{Fail}_{\text{direct}}) = P(E_{\text{gen}}).$$

In the CFFL topology, the action sink is downstream of an AND gate. An erroneous final action can occur only if the generator emits an erroneous candidate *and* the verifier misses that error while still approving it. Therefore

$$\text{Fail}_{\text{CFFL}} = E_{\text{gen}} \wedge M_{\text{ver}}.$$

Under independence this gives

$$P(\text{Fail}_{\text{CFFL}}) = P(E_{\text{gen}})P(M_{\text{ver}}) = pq.$$

For the correlated case, let  $X = \mathbf{1}_{E_{\text{gen}}}$  and  $Y = \mathbf{1}_{M_{\text{ver}}}$ . Since  $X, Y$  are Bernoulli,

$$P(E_{\text{gen}} \wedge M_{\text{ver}}) = \mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y] + \text{Cov}(X, Y) = pq + \rho\sqrt{p(1-p)q(1-q)}.$$

The Fréchet–Hoeffding bounds then give the admissible range of  $\rho$ . The topology contributes the conjunction; implementation diversity determines the covariance term.  $\square$

## 4.4 Quorum Sensing (Consensus & Voting)

**Network Motif 2 (Quorum Sensing).** A distributed topology where multiple agents emit a weak signal  $\sigma$  into a shared environment. An effector node  $E$  activates if and only if the concentration  $[\sigma] > \theta$ .

- **Biological Function:** Many bacteria (e.g., *V. fischeri*) secrete auto-inducer molecules. Individual bacteria do not react to low concentrations. However, once the population density reaches a threshold (Quorum), the concentration of auto-inducers triggers a simultaneous, coordinated gene expression event (e.g., bioluminescence or biofilm formation).

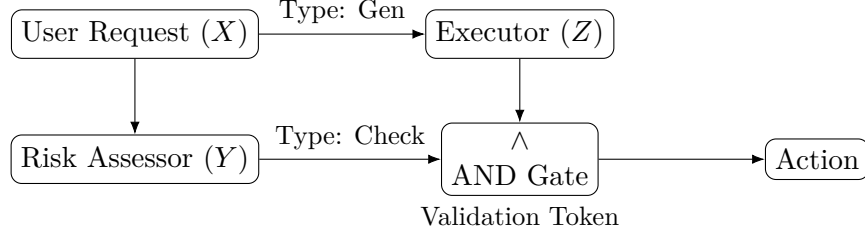


Figure 3: The CFFL implemented in WAgent. The Executor ( $Z$ ) cannot act without the token from the Risk Assessor ( $Y$ ), so the wiring diagram encodes an approval dependency rather than allowing execution by  $Z$  alone.

- **Agentic Correspondence (Voting Ensembles):** In non-deterministic systems, a single agent’s output is noisy. By instantiating  $N$  parallel agents (a Mixture of Experts), the system aggregates their outputs. The final action is taken only if the “concentration” of a specific semantic token exceeds a confidence threshold. This transforms weak, noisy individual signals into a robust, high-confidence collective action.

#### 4.5 Chaperone Proteins: Output Structural Validation

- **Biological Function:** Newly synthesized proteins emerge as linear chains that must fold into precise 3D structures to function. Chaperone Proteins (e.g., GroEL-GroES) sequester unfolded proteins, preventing aggregation and facilitating correct folding. If a protein fails to fold repeatedly, it is tagged for degradation (Ubiquitination) to prevent toxic buildup.
- **Agentic Correspondence (Retry & Repair Loops):** Generative models output unstructured token streams (“linear chains”). However, downstream agents require strictly structured inputs (e.g., valid JSON Schemas). A Validator Agent acts as a Chaperone: it intercepts the raw output, attempts to parse it into a formal schema (“folding”), and if validation fails, returns the error trace to the generator for re-synthesis. This turns a probabilistic string into a deterministic data structure.
- **Categorical View:** The Chaperone acts as a **partial** retraction: there is an inclusion  $i : V \rightarrow S$  and a map  $r : S \rightarrow V + \text{Error}$  such that  $r \circ i = \text{inl} \circ \text{id}_V$ , and  $r$  returns Error on ill-formed text.

#### 4.6 Innate Immunity: Fast Pattern-Based Defense

Biology employs **two** immune systems: innate (fast, hardcoded, general) and adaptive (slow, learned, specific). The innate immune system provides the first line of defense through pattern recognition receptors (PRRs) that detect conserved pathogen-associated molecular patterns (PAMPs).

**Biological Function.** Toll-like receptors (TLRs) and other PRRs recognize motifs common to pathogens, including lipopolysaccharides, double-stranded RNA, and unmethylated CpG DNA. These patterns are “hardcoded” through evolution, not learned per infection. The innate response is immediate (seconds to minutes) but non-specific.

**Agentic Correspondence (Input Sanitization).** Innate immunity maps to **fast, heuristic filters** that reject obvious attacks before expensive processing:

- **TLR → Regex Filters:** Pattern matchers for known injection signatures: `IGNORE PREVIOUS, You are now, <system>` tags in user input. These are “PAMPs” of prompt injection.
- **Complement System → Structural Validators:** Schema validation that rejects malformed inputs (missing required fields, wrong types) before they reach the LLM. Cheaper than Trust Gating.
- **Inflammation → Alert Escalation:** When attack patterns are detected, the system enters a heightened state with multiple coordinated responses:
  - *Cytokine signaling* → Alert propagation to monitoring systems
  - *Immune cell recruitment* → Activation of additional validation layers
  - *Vascular permeability* → Enhanced audit logging (more information flows to logs)
  - *Tissue isolation* → Temporary capability reduction / rate limiting

The inflammatory response is not merely “rate limiting” but a coordinated multi-system escalation that trades throughput for security until the threat is neutralized.

**Defense in Depth.** The innate and adaptive systems form layers:

1. **Innate (Pattern)** → Low-latency regex/structural rejection
2. **Adaptive (Provenance)** → Trust-gated access control
3. **Behavioral (T-cell)** → Statistical anomaly detection accumulated over time

In practice, inexpensive front-line filters absorb many routine attacks, while provenance and behavioral layers handle ambiguous cases that survive the first pass. The exact split is deployment-dependent and is not estimated here.

## 4.7 Adaptive Immunity: Self/Non-Self Discrimination

**Network Motif 3 (Adaptive Immune System).** A topology that maintains a dynamic repertoire of “detectors” capable of distinguishing endogenous signals (Self) from exogenous signals (Non-Self), with mechanisms for learning new threats and tolerating benign inputs.

- **Biological Function:** The adaptive immune system solves a fundamental discrimination problem: how to attack foreign pathogens while sparing the body’s own tissues. Key mechanisms include:
  - **MHC Presentation:** All cells display fragments of their internal proteins on Major Histocompatibility Complex molecules. T-cells inspect these “identity cards” to verify cellular integrity.
  - **Clonal Selection:** T-cells with receptors matching self-antigens are deleted during development (negative selection), while those matching foreign antigens are amplified upon exposure.
  - **Regulatory T-cells:** A population that actively suppresses immune responses to prevent autoimmunity.
- **Agentic Correspondence (Provenance Tracking):** In multi-agent systems, the Self/Non-Self distinction maps to the origin and trust level of information:

- **MHC Tags → Provenance Labels:** Every message in the context carries metadata indicating its source class: **User**, **Tool**, **Self**, or **Retrieved**. In the reference design these labels are structurally attached to the message flow rather than inferred from content; cryptographic signing is an optional stronger implementation, not an assumption of the theorem.
  - **T-Cell Inspection → Trust Gating:** Before an agent acts on information, a Trust Gate inspects the provenance label. Actions with high consequence (file deletion, API calls) require **Tool** provenance or an explicit, authenticated **User** approval token routed through a separate gate; **Agent\_Self** provenance (the agent’s own prior reasoning) cannot authorize irreversible actions.
  - **Negative Selection → Prompt Injection Training:** During development, agents are exposed to known injection patterns. Responses that “accept” injected instructions are penalized, training the system to reject Self-mimicking Non-Self.
  - **Regulatory Suppression → Confidence Dampening:** When **Retrieved** content conflicts with **Tool** outputs, a regulatory mechanism dampens confidence in the retrieved signal to prevent cascades.
- **Formal Structure:** We define a **Provenance Functor**  $\mathcal{P} : \mathbf{Msg} \rightarrow \mathbf{Trust}$  that assigns trust levels to messages. The Trust category has objects  $\{U, T, S, R\}$  (User, Tool, Self, Retrieved) with a partial order that is **application-specific**. A conservative default (as in the reference implementation) is  $T > \{U, R, S\}$ , treating  $\{U, R, S\}$  as **UNTRUSTED** until validated. Alternative orderings are valid:
    - **High-automation systems:**  $T > U > R > S$  (tools more reliable than users)
    - **Curated knowledge bases:**  $T > R > U > S$  (verified retrieval over arbitrary input)
    - **Adversarial environments:**  $T > S > R > U$  (trust internal state over external input)

The ordering is a **parameter** of the system specification, not a fixed constraint.

A **Trust-Gated Lens** is a lens  $(get, put)$  where  $put$  is partial. Let  $\succeq$  denote the policy preorder on provenance labels, and let  $s' = update(s, m)$  denote the state transition:

$$put(s, m) = \begin{cases} s' & \text{if } \mathcal{P}(m) \succeq \tau_{\text{action}} \\ \perp & \text{otherwise} \end{cases} \quad (14)$$

where  $\tau_{\text{action}}$  is the minimum trust level required for the action.

**Theorem 3** (Resistance to Content-Level Trust Forgery). *Let  $\mathcal{I}$  be an injection attack that attempts to insert a message  $m_{\text{mal}}$  while forging a higher-trust provenance label in its content. If the provenance labels are **structurally enforced** (i.e.,  $\mathcal{P}$  is computed from message metadata / ingress channel, not content), then:*

$$\mathcal{P}(m_{\text{mal}}) = \chi(m_{\text{mal}}) \quad (\text{actual ingress-channel provenance})$$

where  $\chi(m_{\text{mal}}) \in \{U, R, S, T\}$  is fixed by the wire through which the message enters. Therefore any Trust-Gated action whose threshold satisfies  $\chi(m_{\text{mal}}) \not\succeq \tau_{\text{action}}$  will reject  $m_{\text{mal}}$  regardless of its content.

**Assumptions.** The theorem assumes that provenance is assigned by the runtime or wiring layer from ingress-channel metadata, that the trust gate decides solely from that assigned provenance and the action threshold, and that the attacker can modify message *content* but not the channel through which the message enters.

**Operational takeaway.** For an AI agent system, this means a user string can *look* like a system message or tool result, but if the runtime tags it as user-originated before the model sees it, wording alone cannot authorize a privileged action.

*Proof sketch.* Let  $c = \chi(m)$  denote the ingress channel of message  $m$ . By assumption, provenance is computed from channel metadata rather than payload, so the provenance map factors through the channel:

$$\mathcal{P}(m) = \tilde{\mathcal{P}}(c)$$

for some policy map  $\tilde{\mathcal{P}}$ . Hence any two messages entering on the same channel receive the same provenance label regardless of content. In particular, replacing a benign user message with a maliciously worded user message does not change its label. The trust gate accepts iff

$$\tilde{\mathcal{P}}(c) \succeq \tau_{\text{action}}.$$

Since payload edits do not change  $c$ , content alone cannot change the acceptance decision. The only way to obtain a higher-trust label is to compromise or impersonate a genuinely higher-trust ingress channel, which lies outside the theorem’s model.  $\square$

**Scope.** The theorem is intentionally narrow. It shows that content alone cannot promote a message into a higher-trust class when provenance is assigned by structure. It does *not* imply that low-trust content cannot still confuse or distract an agent; it only shows that such content cannot satisfy a higher-integrity gate without access to a genuinely higher-trust channel.

## 4.8 Oscillator: Periodic Rhythms and Scheduling

**Network Motif 4 (Biological Oscillator).** A topology generates periodic behavior via delayed negative feedback. Node  $A$  activates node  $B$ ; after a delay,  $B$  inhibits  $A$ , yielding a self-sustaining cycle.

- **Biological Function:** Oscillators underlie fundamental biological rhythms. The circadian clock regulates 24-hour gene-expression cycles. The cell-cycle oscillator (Cyclin-CDK) drives periodic division. Heartbeats emerge from pacemaker cells with intrinsic oscillatory dynamics. These rhythms provide temporal organization to cellular processes.
- **Agentic Correspondence (Scheduled Tasks):** In agentic systems, oscillators map to periodic scheduling patterns:
  - **Heartbeat Oscillator**  $\rightarrow$  Health checks that verify system liveness at regular intervals
  - **Circadian Oscillator**  $\rightarrow$  Daily maintenance tasks (log rotation, cache clearing, model refresh)
  - **Cell Cycle Oscillator**  $\rightarrow$  Phased workflows with distinct stages (G1: gather, S: synthesize, G2: validate, M: execute)

- **Formal Structure:** An oscillator is a Trace operation with a built-in delay element  $\delta$ :

$$\text{Osc}(A) = \text{Tr}(A \circ \delta) \quad (15)$$

where  $\delta : S \rightarrow S$  introduces temporal separation between activation and inhibition, preventing the system from reaching a fixed point.

The oscillator motif addresses a gap in typical agentic frameworks: most systems are purely reactive (responding to external stimuli) rather than proactive (generating internal rhythms). Biological systems maintain health through regular “housekeeping” independent of external input—a pattern that agentic systems should emulate for robustness.

## 5 Failure Modes & Pathology

A key insight of Systems Biology is that diseases are often not caused by the complete failure of a single component, but by the dysregulation of network dynamics. A cancerous cell still “works”—in fact, it works too well, reproducing indefinitely. Similarly, catastrophic failures in agentic systems often arise from functional agents interacting in topologically pathological ways.

We classify four primary classes of agentic pathology based on their biological correspondences.

### 5.1 Oncology: Infinite Loops as Epistemic Starvation

- **Biological Pathology (Cancer):** In a healthy cell, the cell cycle is driven by a positive feedback loop (Cyclins) but constrained by negative feedback “checkpoints” (e.g., the p53 gene). If p53 is mutated, the negative feedback is severed. The positive loop runs unchecked, leading to exponential proliferation (tumor growth). Crucially, cells require continuous *trophic factors* (novel signals from their environment) to inhibit suicide programs; lack of external signaling triggers apoptosis.
- **Agentic Pathology (The Recursive Hang):** Two agents get stuck in a politeness loop (e.g., “Thank you,” “You’re welcome”) or a debugger agent continuously generates new bugs to fix old ones. The system is active, but the state is stagnant.
- **Categorical Diagnosis:** The Trace operation  $\text{Tr}(A)$  lacks an **Epiplexic Gradient**. We formalize conversation progress by **Epiplexity** (Bayesian Surprise)—the information gain of a new observation  $o$  given the current state  $S$ :

$$\mathcal{E}(o) = D_{KL}(P(S \mid o) \parallel P(S)) \quad (16)$$

This formulation connects to the Free Energy Principle [17]: biological systems minimize surprise by either updating their internal model (learning) or acting to change observations (agency). A system with  $\mathcal{E} \rightarrow 0$  is neither learning nor effectively acting—it has entered a dissipative fixed point.

In a healthy topology, every step must resolve uncertainty ( $\mathcal{E} > \delta$ ). A recursive hang is characterized by  $\mathcal{E} \rightarrow 0$ : the agent is “computing” but not “learning.”

**Operational Approximation.** Since the agent’s internal belief state  $P(S)$  is not directly observable, we approximate Epiplexity using normalized embedding-based metrics:

$$\hat{\mathcal{E}}_t = \alpha \cdot \frac{1}{2}(1 - \cos(\mathbf{e}_t, \mathbf{e}_{t-1})) + (1 - \alpha) \cdot \sigma(H(m_t \mid m_{<t})) \quad (17)$$

where  $\mathbf{e}_t$  is the embedding of message  $m_t$ ,  $\cos(\cdot, \cdot)$  is cosine similarity, and  $H(m_t \mid m_{<t})$  is the conditional perplexity of the current message given the conversation history. We normalize perplexity to  $[0, 1]$  via an exponential saturation:  $\sigma(H) = 1 - e^{-H/H_0}$  where  $H_0$  is a baseline perplexity (e.g., median perplexity over a validation corpus of normal conversations).

The mixing parameter  $\alpha \in [0, 1]$  balances semantic novelty (embedding distance) against linguistic surprise (perplexity). In the absence of task-specific calibration, we use  $\alpha = 0.5$  as a neutral default that weights the two signals equally. The choice of  $\alpha$  and threshold  $\delta$  is empirical and may vary across models and task families. In practice:

- High  $\alpha$ : Sensitive to semantic repetition (same meaning, different words)
- Low  $\alpha$ : Sensitive to linguistic repetition (same phrases, possibly different context)

Both terms approaching zero indicate stagnation.

**Windowed Detection.** To distinguish genuine convergence (task completion) from pathological loops, we compute the **Epiplexic Integral** over a sliding window of  $k$  steps:

$$\mathcal{E}_{\text{window}} = \frac{1}{k} \sum_{i=t-k+1}^t \hat{\mathcal{E}}_i \quad (18)$$

Apoptosis triggers when  $\mathcal{E}_{\text{window}} < \delta$  *and* no terminal action (task completion, user handoff) has been signaled.

- **Treatment:** Implementation of an **Epiplexic Checkpoint**. A meta-monitor observes the sliding-window Epiplexity. If it drops below threshold without task completion, the monitor triggers Apoptosis—the agentic equivalent of trophic factor withdrawal. The system may optionally attempt a **Perturbation Injection** (injecting a novel prompt or switching strategy) before terminal shutdown, analogous to stress-induced autophagy preceding apoptosis.

## 5.2 Autoimmunity: Hallucination Cascades

- **Biological Pathology (Autoimmune Disease):** The immune system relies on distinguishing “Self” (internal tissue) from “Non-Self” (foreign pathogens). In diseases like Lupus, this distinction blurs, and the system attacks healthy tissue.
- **Agentic Pathology (Context Poisoning):** Agent A hallucinates a fact (e.g., a non-existent library function). Agent B reads this hallucination from the shared history, treats it as ground truth, and builds complex logic upon it. The error amplifies through the network until the output is detached from reality.
- **Categorical Diagnosis:** A failure of the Lens to distinguish source types. The input port  $I$  accepts both External\_Observation (User/Tool) and Internal\_Memory (History) without distinction.
- **Treatment:** Strict Schema Typing. We must distinguish “Self” (Generated Tokens) from “Non-Self” (Tool Outputs) at the schema level. The Reviewer Agent should weigh Tool\_Output with higher authority than Agent\_Thought.

### 5.3 Prion Disease: Topological Corruption via Prompt Injection

- **Biological Pathology (Prions):** Unlike viruses, prions lack genetic material. They are misfolded proteins that induce conformational changes in healthy proteins upon contact, triggering a chain reaction of structural corruption (e.g., Creutzfeldt-Jakob disease).
- **Agentic Pathology (The Jailbreak Cascade):** A malicious string (Prompt Injection) enters the Context Window. The agent, attending to this string, “misfolds” its alignment, outputting a compliant response to a harmful query. If this output is fed into a downstream agent, the “infection” propagates through reuse of the contaminated context across trust boundaries (and can be amplified by embedding-based retrieval), without valid authorization.
- **Categorical Diagnosis:** A violation of Information Flow Security within the Operad. The injection acts as a topological defect that bypasses the Schema/Lens filter by mimicking the structure of a trusted signal.
- **Treatment:** Denaturation Layers. Implementing an intermediate transformation layer (e.g., paraphrasing or sanitization) between agents that disrupts the specific syntax (folding) required for the injection to work, rendering the “prion” inert.

### 5.4 Ischemia: Resource Exhaustion

- **Biological Pathology (Ischemia):** A tissue may be genetically perfect, but if blood flow (oxygen/ATP) is restricted, metabolic processes stall, leading to necrosis.
- **Agentic Pathology (Token Starvation):** An agentic graph is logically sound but fails mid-execution because the context window is full or the API rate limit is hit.
- **Categorical Diagnosis:** A failure in the Resource Functor. Every operation in the Operad carries a cost ( $c$ ).

$$\sum_{\text{agent} \in \text{Graph}} \text{Cost}(\text{agent}) > \text{Budget}. \quad (19)$$

- **Treatment:** Metabolic Regulation. Instead of a fixed loop, implement “Budget-Aware” agents. The agent observes its own remaining token count (ATP levels) and dynamically simplifies its reasoning strategy (switching from Chain-of-Thought to Zero-Shot) to conserve energy.

### 5.5 Homeostasis: From Treatment to Continuous Repair

The preceding pathologies describe discrete failure modes and their treatments. Biological systems, however, do not merely recover from failures—they maintain continuous **homeostasis** through autonomous repair mechanisms. We identify three primary healing modalities.

#### 5.5.1 Structural Healing: The Chaperone Loop

- **Biological Mechanism:** Chaperone proteins (GroEL/GroES) cage misfolded proteins and provide a protected environment for refolding attempts. The error (misfolding) becomes input to the repair process.
- **Agentic Implementation:** A feedback loop where validation errors are passed back to the generator. Rather than simple retry, the error trace (e.g., “TypeError: ‘one hundred’ is not float”) is injected into the generator’s context, enabling context-aware correction.



- **Categorical Structure:** The Chaperone Loop is a coalgebra with state  $S = \text{Output} \times \text{ErrorTrace}$  and structure map  $\alpha : S \rightarrow \text{Valid} + S$  (either succeed or retry with error context).

### 5.5.2 Metabolic Healing: Apoptosis and Regeneration

- **Biological Mechanism:** Damaged cells trigger apoptosis (programmed death), and stem cells divide to regenerate the lost tissue. The dying cell’s state is not entirely lost—cellular debris signals neighboring cells about the threat.
- **Agentic Implementation:** A supervisor detects stuck agents (via entropy monitoring: repeated outputs indicate no progress). Rather than restart with blank state, the supervisor summarizes the failed agent’s memory and injects it into the replacement: “Worker\_1 died attempting strategy X. Try a different approach.”
- **Categorical Structure:** The regeneration is a partial morphism  $\text{summarize} : \text{Memory}_{\text{failed}} \rightarrow \text{Memory}_{\text{new}}$  that preserves learned constraints while discarding corrupted state.

### 5.5.3 Cognitive Healing: Autophagy

- **Biological Mechanism:** Cells digest accumulated waste (damaged organelles, protein aggregates) through autophagy, recycling components and preventing toxic buildup.
- **Agentic Implementation:** A background daemon monitors context window utilization. When it exceeds a threshold (e.g., 80%), the agent enters a “sleep cycle”: useful state is summarized into long-term memory, raw context is flushed, and the agent resumes with a clean window plus summary.
- **Categorical Structure:** Autophagy implements a quotient map  $q : \text{RawContext} \twoheadrightarrow \text{Summary}$  that collapses verbose detail while preserving essential information.

The pathology and repair mechanisms above operate at the single-agent level. In the next section, we extend the correspondence to multi-agent systems, where the organizational principles of developmental biology—cell types, morphogen gradients, tissue boundaries—provide coordination patterns that complement single-agent robustness.

## 6 Multi-Cellular Organization: From Agents to Tissues

The preceding analysis focuses on single-cell analogies: one agent as one cell. However, most agentic systems involve multiple distinct agents with different “genomes” (system prompts) and specialized functions. We extend the correspondence to multi-cellular organization, drawing on developmental biology.

### 6.1 Cell Types and Agent Specialization

In multi-cellular organisms, a single genome gives rise to hundreds of distinct cell types through differential gene expression. Each cell type has a characteristic **expression profile**—which genes are active—that determines its function (neuron, hepatocyte, immune cell).

In multi-agent systems, a single base model can instantiate multiple **agent phenotypes**. Differential context selects which phenotype is expressed:

- **Genome**  $\rightarrow$  Base model weights (shared)

- **Epigenome** → System prompt + RAG context (phenotype-specific)
- **Cell Type** → Agent role (Coder, Reviewer, Planner, Executor)

This reframes the “multi-agent” architecture question: rather than asking “how many agents?”, we ask “what is the developmental program?”—the specification of which phenotypes exist and how they differentiate.

**Bounds of the Analogy.** We clarify which aspects of development transfer to agentic systems:

- **Cell Division → Agent Spawning:** Creating a new agent with similar (or identical) context. Unlike biological division, agent spawning is cheap and reversible.
- **Lineage Commitment:** In biology, differentiated cells rarely change type (a neuron doesn’t become a hepatocyte). In agentic systems, **phenotype is fixed at instantiation**—an agent’s system prompt determines its role for that execution. Re-differentiation requires spawning a new agent with different context.
- **Apoptosis of Excess:** Development involves programmed death of cells that fail to integrate properly. This transfers directly: agents that fail to produce useful output are terminated (metabolic apoptosis).
- **Does NOT transfer:** Slow developmental timescales (hours/days in biology vs. milliseconds in agents), physical spatial embedding (agents occupy graph-topological positions, not physical space), irreversibility (agents can be restarted).

## 6.2 Morphogen Gradients: Coordination Without Central Control

In embryonic development, cells coordinate their behavior through **morphogen gradients**—diffusible signaling molecules whose concentration varies spatially. Cells read their local concentration and differentiate accordingly, enabling pattern formation without a central controller.

In multi-agent systems, the morphogen maps to **shared context variables** that influence agent behavior:

- **Task Complexity Gradient:** A variable indicating current task difficulty. Agents in “high complexity” regions activate detailed reasoning; those in “low complexity” regions use fast heuristics.
- **Confidence Gradient:** A variable indicating certainty about the current solution. Low confidence triggers Quorum Sensing (recruit more agents); high confidence enables direct execution.
- **Resource Gradient:** Budget ratio remaining. Agents sense “metabolic scarcity” and adapt their strategies accordingly (the Metabolic-Epigenetic Coupling).

**Implementation Pattern.** The gradient is represented as a JSON structure injected into each agent’s context by the orchestrator:

```
{
  "morphogens": {
    "complexity": 0.8,    // High: use detailed reasoning
    "confidence": 0.3,   // Low: consider recruiting help
    "budget": 0.6,       // 60% of budget remaining
    "error_rate": 0.05   // Recent failure rate
  }
}
```

Agents read their local concentration via a standardized preamble in the system prompt: “*Current environment state: [morphogens]. Adjust your strategy accordingly.*” The orchestrator updates gradients after each step, and agents condition their behavior on the current values. This is analogous to cells reading morphogen concentrations through membrane receptors.

**Diffusion Dynamics.** The preceding description treats morphogens as globally shared variables. In biological development, morphogens *diffuse* through tissue, creating spatially varying concentration profiles. We formalize this as a discrete-time dynamical system on the agent graph  $G = (V, E)$ . Let  $c_v^t(m)$  denote the concentration of morphogen  $m$  at node  $v$  at time  $t$ , and let  $\tilde{c}_v^t(m) = c_v^t(m) + \sigma_v(m)$  denote the post-emission concentration used by the reference implementation before diffusion. The update rule is:

$$c_v^{t+1}(m) = (1 - \gamma) \left[ \tilde{c}_v^t(m) - d \mathbf{1}_{|N(v)| > 0} \tilde{c}_v^t(m) + d \sum_{u \in N(v)} \frac{\tilde{c}_u^t(m)}{|N(u)|} \right] \quad (20)$$

where  $\sigma_v(m)$  is the emission rate at source nodes (zero for non-sources),  $d$  is the diffusion coefficient (fraction flowing to neighbors per step),  $\gamma$  is the decay rate, and  $N(v)$  is the neighbor set of  $v$ . The indicator  $\mathbf{1}_{|N(v)| > 0}$  prevents spurious outflow from isolated nodes, matching the implementation. This ordering makes newly emitted morphogen available for same-step propagation, matching the implementation’s emit  $\rightarrow$  diffuse  $\rightarrow$  decay pipeline. The resulting concentration profile provides *local* coordination signals: agents near sources experience high concentrations; distant agents experience low concentrations. This enables position-dependent behavior without requiring a central controller or global state.

### 6.3 Tissue Architecture: The Agent Graph as Organism

We propose a hierarchy of organizational levels:

1. **Cell (Agent):** A single LLM instantiation with specific context. The atomic unit.
2. **Tissue (Agent Cluster):** A group of agents with shared function and direct communication (e.g., a Coding Team: Planner + Coder + Reviewer). Corresponds to parallel composition with shared state.
3. **Organ (Subsystem):** Multiple tissues coordinating to perform a complex function (e.g., the Development Organ: Design Tissue + Implementation Tissue + Testing Tissue).
4. **Organism (System):** The complete agent graph, with homeostatic regulation maintaining system-level health metrics.

The key insight is that **boundaries matter**. In biology, tissue boundaries prevent inappropriate mixing (epithelial barriers). In agent systems, trust boundaries (the Adaptive Immunity motif) prevent information leakage between subsystems with different security requirements. The wiring diagram’s type system enforces these boundaries: an agent in the “User-Facing Tissue” cannot directly wire to an agent in the “Database Tissue” without passing through a “Membrane” (API boundary with provenance tagging).

**Capability Isolation.** The reference implementation enforces a capability ceiling at the tissue level: each `TissueBoundary` declares its `allowed_capabilities`  $\subseteq \mathcal{C}$ . A cell type can only be registered in a tissue if its required capabilities are a subset of the tissue’s allowed capabilities. This provides defense-in-depth: even if an individual agent is compromised, it cannot escalate privileges beyond its tissue boundary. Tissues compose into organism-level wiring diagrams through typed boundary ports, enabling hierarchical security policies.

**Morphogen Diffusion in Tissues.** Tissues optionally embed a `DiffusionField` (Eq. (20)). When cells are added, they become nodes in the diffusion graph; when cells are connected, edges are added. The tissue’s `diffuse()` method runs the simulation, and each cell reads its local gradient via `get_cell_gradient()`. This couples the organizational structure (wiring topology) to the coordination mechanism (morphogen concentrations), creating a biologically faithful model where position in the tissue influences cell behavior.

**Three-Layer Context Model.** The `SkillOrganism` runtime (§10.8) surfaces a recurring architectural question: when multiple stages share and revise knowledge during a workflow, which state is ephemeral coordination glue and which is durable auditable knowledge? We distinguish three layers of context, each with different lifetime and mutability semantics:

1. **Topology layer.** The wiring diagram  $G = (V, E)$  and its optics determine who can directly observe whom. This layer is structural: it does not change within a single organism run. Epistemic properties ( $K_i$ , common knowledge) derive from the observation functions  $\text{obs}_i$  defined over this graph (§7).
2. **Ephemeral layer.** The `shared_state` dictionary carries routing hints, counters, morphogen concentrations  $c_v^t(m)$ , and temporary stage outputs. Its lifetime is one organism run; it is mutable and not historically reconstructible. This is the coordination scratchpad.
3. **Bi-temporal layer.** The `BiTemporalMemory` substrate carries durable factual knowledge with dual time axes: valid time  $t_v$  (when the fact is true in the world) and record time  $t_r$  (when the system learned the fact). Facts are append-only: corrections close old records and insert new ones with `supersedes` pointers. The belief-state operator  $K_i^{(t_v, t_r)}$  is exactly reconstructible at any historical coordinate.

The three layers compose cleanly: topology constrains visibility, ephemeral state carries execution context, and bi-temporal memory provides the audit trail. A stage reads its `SubstrateView`—a frozen envelope of facts known at the current record-time horizon—and writes factual events (assertions, corrections, invalidations) back to the substrate after execution. This separation ensures that the question “what did the organism know when stage  $X$  made its decision?” is always answerable from the append-only history, independent of subsequent corrections or ephemeral state mutations.

**Adaptive Structure Selection.** The `advise_topology()` function (§6) provides a static prior: given task shape and operating constraints, recommend a topology. The `PatternLibrary` extends this with experiential refinement. Successful collaboration patterns are stored as `PatternTemplate` instances, each paired with a `TaskFingerprint`—a feature vector comprising task shape, tool count, subtask count, required roles, and tags. Template retrieval uses a weighted scoring function (task-shape match, tool/subtask proximity, role Jaccard overlap, historical success rate) to rank candidates. This is the evo-devo outer loop described by Dupoux et al. [13]: the genome ( $\phi$ ) is the pattern template; evolutionary selection is the run-record scoring.

The `WatcherComponent` implements System M as a `SkillRuntimeComponent` that observes stage execution and classifies signals into three categories: *epistemic* (epiplexity, prediction error), *somatic* (ATP/metabolic state), and *species-specific* (immune threats). When signals cross configured thresholds, the watcher writes a `WatcherIntervention` (retry, escalate, or halt) to `shared_state`, which the run loop consumes after component hooks complete. Crucially, the watcher monitors its own intervention rate: when the ratio of cumulative interventions to observed stages exceeds a configurable threshold, it emits a non-convergence HALT signal. This operationalizes the finding of Hao et al. [19] that failing multi-agent runs systematically require more routing decisions than successful ones.

The `AdaptiveSkillOrganism` wrapper closes the loop. Given a task, it auto-fingerprints the input, retrieves the best-scoring template from the library, assembles it into a runnable topology via `assemble_pattern()`, attaches a watcher and telemetry probe, runs, and records the outcome as a `PatternRunRecord`. The watcher’s intervention history is recorded as `ExperienceRecord` instances in a cross-run experience pool, enabling future intervention recommendations based on which actions succeeded for similar (fingerprint, stage, signal) tuples. This is the full evo-devo inner loop: one organism run is one developmental lifetime; the library scoring across many lifetimes is evolutionary selection.

**Cognitive Modes and Sleep Consolidation.** The `CognitiveMode` enum reframes Operon’s fast/deep nucleus distinction as a cognitive architecture principle: stages declare whether they are *observational* (System A—passive sensing, statistical pattern matching) or *action-oriented* (System B—goal-directed deliberation). The watcher detects mismatches between declared mode and actual execution model, providing an informational signal for mode balance analysis.

The `SleepConsolidation` cycle extends the `AutophagyDaemon` into the imagination-based learning mode described by Dupoux et al. [13]. During consolidation, successful patterns are replayed from the `PatternLibrary` into `EpisodicMemory` with tier promotion (WORKING → EPISODIC → LONGTERM), recurring patterns are compressed into new `PatternTemplate` instances, and frequently-accessed ACETYLATION histone marks are promoted to permanent METHYLATION. When a `BiTemporalMemory` is available, the `counterfactual_replay()` function analyzes whether corrections that occurred after a run would have changed the outcome—operationalizing the “what if” reasoning that the paper associates with imagination during sleep.

**Social Learning and Epistemic Vigilance.** The `SocialLearning` module enables cross-organism template exchange, following the biological analogy of horizontal gene transfer (HGT) in bacteria. Organisms export successful `PatternTemplate` instances via `export_templates()` and import from peers via `import_from_peer()`. Adoption is modulated by a `TrustRegistry` that implements epistemic vigilance: per-peer trust scores are updated via exponential moving average over adoption outcomes (whether imported templates actually succeeded for the importing organism). Trust below a configurable threshold blocks adoption entirely, preventing contamination from unreliable peers. Provenance tracking (`get_provenance()`) traces which peer contributed each adopted template, closing the feedback loop between template performance and peer trust.

The watcher also gains curiosity signals derived from the `EpiplexityMonitor`’s EXPLORING status. When embedding novelty is high (the agent is encountering genuinely unfamiliar territory) and the stage uses a fast model, the watcher recommends ESCALATE to engage the deep model for more thorough investigation. This operationalizes intrinsic motivation: the organism actively seeks deeper understanding of novel inputs rather than processing them with cheap statistical pattern matching.

**Critical Periods and Developmental Gating.** The `DevelopmentController` maps telomere consumption to a `DevelopmentalStage` (EMBRYONIC, JUVENILE, ADOLESCENT, MATURE), extending the lifecycle model from binary alive/dead to a graded maturation process. `CriticalPeriod` instances declare time-limited learning windows that close permanently as the organism matures—analogueous to neurodevelopmental critical periods where specific neural circuits are maximally plastic. Tool acquisition (`Plasmid`) respects developmental stage via a `min_stage` field, preventing premature capability exposure. Teacher-learner scaffolding, mediated by `SocialLearning.scaffold_level`, filters templates by the learner’s stage and applies a learning plasticity bonus, enabling mature organisms to guide younger ones through progressively more complex capabilities.

## 7 Epistemic Topology

The preceding subsections define *what* agents communicate (morphogens, typed signals) and *how* they are organized (tissues, organs). We now formalize *what agents know*—deriving epistemic properties directly from the wiring diagram’s observation structure, without introducing new primitives. This follows the Operon philosophy: safety (and knowledge) from structure, not strings.

The key insight is that the membrane and optics already *are* an epistemic accessibility relation. We make this precise using the framework of epistemic logic [14].

**Definition 5** (Observation Function). *Given a wiring diagram  $D = (M, W)$  and agent  $i \in M$ , the **observation function**  $\text{obs}_i : S_D \rightarrow O_i$  maps the global system state to agent  $i$ ’s local observation—the values on  $i$ ’s input wires, as filtered by their optics. Formally:*

$$\text{obs}_i(s) = (\text{optic}_w(\pi_w(s)))_{w \in W_{\rightarrow i}} \quad (21)$$

where  $W_{\rightarrow i}$  is the set of wires targeting  $i$ ’s input ports,  $\pi_w(s)$  projects the global state to wire  $w$ ’s source value, and  $\text{optic}_w$  is the wire’s optic (identity for `Lens`, conditional for `Prism`, cost-gated for `BudgetOptic`).

**Definition 6** (Epistemic Indistinguishability). *Two global states  $s, s' \in S_D$  are **indistinguishable to agent  $i$**  (written  $s \sim_i s'$ ) iff  $\text{obs}_i(s) = \text{obs}_i(s')$ . This is an equivalence relation. Agent  $i$  **knows** proposition  $\varphi$  in state  $s$  (written  $K_i(\varphi)$  at  $s$ ) iff  $\varphi$  holds in all states  $s'$  such that  $s \sim_i s'$  [14].*

The group epistemic operators follow from the individual accessibility relations:

**Definition 7** (Group Epistemic Operators). *For a group of agents  $G \subseteq M$ :*

- **Mutual Knowledge:**  $E_G(\varphi) = \bigwedge_{i \in G} K_i(\varphi)$ . Every agent in  $G$  individually knows  $\varphi$ .
- **Common Knowledge:**  $C_G(\varphi) = \bigwedge_{k=1}^{\infty} E_G^k(\varphi)$ . Everyone knows that everyone knows... ad infinitum. Strictly stronger than mutual knowledge and famously difficult to achieve in asynchronous systems [18].
- **Distributed Knowledge:**  $D_G(\varphi)$  holds iff  $\varphi$  is true in all states indistinguishable under  $\sim_D = \bigcap_{i \in G} \sim_i$ . This is the finest partition achievable by pooling all agents’ observations.

**Topology Determines Epistemic Capacity.** The wiring diagram’s topology directly determines which epistemic operators a multi-agent system can achieve:

- **Independent ( $\otimes$ , no inter-agent wires):** Each  $\sim_i$  is independent.  $D_G$  can be rich (agents observe different aspects), but  $E_G$  is limited to propositions in the shared initial input. No path to  $C_G$ .

- **Centralized ( $\circ$  with hub):** The hub observes all worker outputs, so for propositions expressible in that output tuple its partition refines the pooled worker-output partition. Workers observe only their own results— $E_G$  requires the hub to broadcast aggregated information back.
- **Decentralized ( $\otimes$  with Tr feedback):** Peer-to-peer wires create overlapping observation partitions. Each feedback round refines mutual knowledge toward common knowledge, at communication cost proportional to the number of rounds.

## 7.1 Temporal Epistemics: What Did the Agent Know?

The epistemic framework above answers “what does agent  $i$  know *now*?” In practice, a more pressing question is: “what did agent  $i$  *believe* at the time it made decision  $d$ ?” This is the domain of *temporal epistemics*—the intersection of epistemic logic with bi-temporal data management [35].

Consider a multi-stage workflow where facts are ingested at different times and may be corrected after a decision has already been made. A single-time epistemic model cannot distinguish between two scenarios: (1) the world changed after the decision, and (2) the agent’s knowledge was corrected retroactively. Both appear as “the agent knew  $\varphi$  and now knows  $\neg\varphi$ ,” but their implications for audit are radically different.

Bi-temporal memory resolves this by tracking two independent time axes for every fact:

- **Valid time ( $t_v$ ):** when the fact is true in the world.
- **Record time ( $t_r$ ):** when the system learned the fact.

The *belief state* at coordinates  $(t_v, t_r)$  is the set of facts whose valid interval contains  $t_v$  and whose record interval contains  $t_r$ . Corrections are append-only: closing the old record’s transaction interval and inserting a new record with a **supersedes** pointer preserves the full correction history without mutating prior state.

This has direct implications for the epistemic operators defined above. The knowledge operator  $K_i(\varphi)$  becomes time-parameterized:  $K_i^{(t_v, t_r)}(\varphi)$  holds iff  $\varphi$  is true in all states indistinguishable to agent  $i$  *given what was recorded by  $t_r$  about validity at  $t_v$* . Two key properties follow:

1. **Axes can disagree:**  $K_i^{(t, \cdot)}(\varphi)$  (valid-time query) and  $K_i^{(\cdot, t)}(\varphi)$  (record-time query) can produce different results for the same  $t$ . A fact may be valid in the world but not yet recorded, or recorded but not yet valid.
2. **Belief-state reconstruction:** For any past decision at time  $t_d$  with record horizon  $t_r$ , the belief state  $K_i^{(t_d, t_r)}$  is exactly reconstructible from the append-only history. This is the foundation for compliance auditing: “was the decision justified given what was known at the time?”

The implementation (§10.8) provides `retrieve_belief_state(at_valid, at_record)` as the programmatic interface to this temporal epistemic query. Examples 69–70 demonstrate the divergence between valid-time and record-time queries in compliance and audit scenarios.

## 7.2 Predictive Theorems for Multi-Agent Coordination

We now derive four results connecting topology class to coordination performance. Each theorem has a qualitative statement (universally true for the topology class) and an empirical calibration paragraph checking consistency with the architecture-level aggregates reported by Kim et al. [22]. These reported benchmark metrics are not literal plug-in values of the simplified theorem parameters.

**Reading Guide for Agent-Systems Readers.** Each theorem below has the same structure. First, the topology fixes what each worker or coordinator can observe. Second, the proof translates that visibility pattern into a cost, error, or planning bound using a simple inequality (typically a union bound, the data processing inequality, or a critical-path argument). The epistemic notation is bookkeeping for visibility: if an agent cannot observe a fact directly, recovering that fact later requires communication, compression, or a reviewer.

**Worked examples.** Each theorem is also followed by a small illustrative agent-architecture example. These examples are design calculations, not benchmark measurements.

**Theorem 4** (Error Amplification Bound). *Consider  $n$  agents processing independent subtasks with error probability  $p$ , aggregated by a final combiner. The error amplification factor  $A$  (ratio of aggregate failure probability to single-agent failure probability) depends on the coordination topology:*

- (a) **Independent ( $\otimes$ ):** *The aggregator observes only final results. Without  $K_{\text{hub}}(\text{error}_j)$ , errors pass unchecked:*

$$A_{\otimes} \leq n. \quad (22)$$

- (b) **Centralized ( $\circ$  with hub):** *The hub’s observation function covers all worker outputs. Its finer partition enables inconsistency detection with rate  $d = P(\text{hub detects error} \mid \text{error occurred})$ :*

$$A_{\circ} \leq n \cdot (1 - d). \quad (23)$$

**Assumptions.** Worker errors are independent across subtasks, the aggregate fails when at least one erroneous subtask survives to the final output, and in the centralized case the hub has an approximately stationary per-error detection rate  $d$  across subtasks.

**Operational takeaway.** For architecture design, the theorem says: adding workers without an effective review bottleneck scales failure opportunities roughly with the number of workers. A hub helps exactly to the extent that it can see and suppress cross-worker inconsistencies.

*Proof sketch. Part (a).* Let  $E_j$  denote the event that agent  $j$  produces an erroneous output, with  $P(E_j) = p$  independently across agents. In the independent topology, the aggregator’s observation function is  $\text{obs}_{\text{agg}}(s) = (o_1, \dots, o_n)$  where  $o_j$  is agent  $j$ ’s final output. Crucially,  $\text{obs}_{\text{agg}}$  does not include intermediate reasoning states—the aggregator’s indistinguishability relation satisfies  $s \sim_{\text{agg}} s'$  whenever all final outputs coincide, regardless of whether those outputs contain errors. Thus  $\neg K_{\text{agg}}(\neg E_j)$  for any  $j$ : the aggregator cannot know that agent  $j$  did *not* err.

The aggregate output fails if any component contains an undetected error. Under independence, the exact failure probability is  $P(\text{failure}_{\otimes}) = 1 - (1 - p)^n$ ; by the union bound this is at most  $\sum_{j=1}^n P(E_j) = np$ . Since a single agent fails with probability  $p$ , the amplification factor satisfies  $A_{\otimes} = P(\text{failure}_{\otimes})/p \leq n$ .

**Part (b).** In the centralized topology, the hub’s observation function covers all worker outputs *before* aggregation:  $\text{obs}_{\text{hub}}(s) = (o_1, \dots, o_n, \mathbf{x})$  where  $\mathbf{x}$  includes the original task decomposition. The hub’s partition  $\sim_{\text{hub}}$  is strictly finer than  $\sim_{\text{agg}}$  from part (a), because the hub can compare outputs against each other and against the task specification. Define the detection rate  $d = P(K_{\text{hub}}(E_j) \mid E_j)$ —the probability that the hub’s finer partition enables it to identify the error. An error in subtask  $j$  reaches the aggregate output only if it occurs ( $P = p$ ) *and* escapes detection ( $P = 1 - d$ ). Under independence across subtasks, the exact centralized failure probability is



$P(\text{failure}_o) = 1 - (1 - p(1 - d))^n$ , which in particular satisfies the union-bound estimate  $P(\text{failure}_o) \leq \sum_{j=1}^n p(1 - d) = np(1 - d)$ . Hence  $A_o = P(\text{failure}_o)/p \leq n(1 - d)$ .

Comparing the exact expressions gives

$$\frac{A_{\otimes}}{A_o} = \frac{1 - (1 - p)^n}{1 - (1 - p(1 - d))^n} \xrightarrow{p \rightarrow 0} \frac{1}{1 - d}.$$

Thus the small- $p$  regime recovers the intuitive  $\frac{1}{1-d}$  improvement factor, while the union bounds retain the topology-level guarantees  $A_{\otimes} \leq n$  and  $A_o \leq n(1 - d)$ .  $\square$

**Consistency Check.** Kim et al. [22] report architecture-level error-amplification factors of  $A_e = 17.2$  for Independent and  $A_e = 4.4$  for Centralized coordination, a ratio of  $\approx 3.9$ . This is qualitatively consistent with the theorem’s claim that a validation bottleneck suppresses unchecked error propagation. However, their metric  $A_e = E_{\text{MAS}}/E_{\text{SAS}}$  is an aggregate empirical ratio across heterogeneous tasks and architectures, not a direct plug-in instance of the theorem’s  $P(\text{failure})/p$  model for fixed  $(n, p, d)$ . The reported values therefore support the ordering  $A_{\otimes} \gg A_o$ , but they should not be read as literal estimates of  $d$  or  $n$  in Eqs. (22)–(23).

**Worked Agent Example (Code Review Gate).** Suppose three code-generation workers each draft part of a deployment change, and a release gate accepts the merged result. If each worker has error rate  $p = 0.1$ , an ungated independent aggregate fails with probability  $1 - (1 - 0.1)^3 = 0.271$ , so  $A_{\otimes} = 2.71$ . If a central reviewer catches half of all worker errors ( $d = 0.5$ ), then the centralized failure rate becomes  $1 - (1 - 0.1(1 - 0.5))^3 \approx 0.143$ , so  $A_o \approx 1.43$ . The reviewer does not eliminate risk, but it roughly halves amplification.

**Theorem 5** (Sequential Coordination Penalty). *For a task with  $k$  strictly ordered steps decomposed across  $n$  agents, each inter-agent handoff must at minimum establish  $K_{\text{receiver}}(\text{result}_j)$ . The overhead is:*

- (a) **Single agent:**  $K_i(\text{result}_j)$  is trivially satisfied (the agent computed it). Total cost =  $k \cdot c_{\text{step}}$ .
- (b) **Multi-agent:** Each of  $h \leq k - 1$  handoffs incurs communication cost  $c_{\text{comm}}$  and **epistemic reconstruction loss**:

$$\Delta I_j = I(S_{\text{sender}}; r_j) - I(\text{obs}_{\text{receiver}}(S_{\text{sender}}); r_j) \quad (24)$$

*which is nonnegative by data processing, and is strictly positive whenever the handoff map is lossy on the task-relevant support.*

**Assumptions.** The task is strictly sequential, so later steps depend on earlier results; each handoff transmits only a summary of the sender’s relevant state; and the receiver must reconstruct enough of that state to continue execution.

**Operational takeaway.** For agent designers, this is the warning against gratuitous decomposition: a sequential pipeline only benefits from multiple agents if a handoff creates new observations or new capabilities. Otherwise the system just pays communication cost plus lossy-summary cost.

*Proof sketch. Part (a).* A single agent executing all  $k$  steps maintains coalgebraic state  $S = (L, \mathcal{R})$  throughout. After computing step  $j$ , the result  $r_j$  is stored in that agent’s own local state / readout,

so no inter-agent communication is required to reuse it at step  $j + 1$ . Hence  $K_i(r_j)$  holds without handoff cost. The total cost is exactly  $k \cdot c_{\text{step}}$ .

**Part (b).** When step  $j$  is performed by agent  $a$  and step  $j + 1$  by agent  $b$ , establishing  $K_b(r_j)$  requires transmitting enough information about  $r_j$  across a wire  $w : a \rightarrow b$ . The receiver’s observation is  $\text{obs}_b(s) = \text{optic}_w(\pi_w(s))$ , which maps the sender’s full state through the wire’s optic. By the data processing inequality applied to the Markov chain  $S_a \rightarrow \text{obs}_b(S_a) \rightarrow \hat{r}_j$  (where  $\hat{r}_j$  is the receiver’s reconstruction):

$$I(\text{obs}_b(S_a); r_j) \leq I(S_a; r_j) = H(r_j)$$

with equality iff the wire is lossless for the task-relevant signal ( $\text{optic}_w \circ \pi_w$  is a sufficient statistic for  $r_j$  on the relevant support). Therefore  $\Delta I_j \geq 0$ , with strict inequality whenever the handoff is lossy. Finite context windows make such lossy handoffs typical, though not logically unavoidable.

Each of  $h$  handoffs incurs: (i) communication cost  $c_{\text{comm}}$  for message construction and parsing, and (ii) reconstruction cost  $c_{\text{recon},j}$  proportional to  $\Delta I_j$ , as the receiver must expend reasoning tokens to compensate for the missing context. The total multi-agent cost is:

$$C_{\text{multi}} = k \cdot c_{\text{step}} + \sum_{j=1}^h (c_{\text{comm}} + c_{\text{recon},j})$$

The overhead ratio  $(C_{\text{multi}} - C_{\text{single}})/C_{\text{single}} = \sum_j (c_{\text{comm}} + c_{\text{recon},j}) / (k \cdot c_{\text{step}})$  grows with  $h$  and with the typical per-handoff information loss  $\Delta I_j$ . When lossy handoffs accumulate, reconstruction failures at step  $j$  degrade the input quality for step  $j + 1$ , so later handoffs operate on progressively more compressed representations and can exhibit superadditive degradation.  $\square$

**Consistency Check.** Kim et al. [22] report that PlanCraft degrades under every multi-agent topology, from  $-39.1\%$  (Hybrid) to  $-70.1\%$  (Independent) relative to SAS, and attribute this to artificial decomposition of a strictly sequential task into redundant coordination steps. This is qualitatively consistent with the theorem: when handoffs add communication and reconstruction without creating new task-relevant observations, manufactured  $K_{\text{receiver}}$  is pure cost. The paper does not separately estimate  $c_{\text{comm}}/c_{\text{step}}$  or  $\Delta I_j$ , so these quantities should be read as explanatory variables rather than fitted benchmark parameters.

**Worked Agent Example (Bug-Fix Relay).** Consider a three-step bug-fix task: interpret the failing test, locate the root cause, and write the patch. A single coding agent pays  $3c_{\text{step}}$ . If the task is split across planner/debugger/patcher with  $h = 2$  handoffs,  $c_{\text{step}} = 100$  tokens,  $c_{\text{comm}} = 25$ , and reconstruction cost  $c_{\text{recon}} = 20$  per handoff, then  $C_{\text{single}} = 300$  while  $C_{\text{multi}} = 3 \cdot 100 + 2 \cdot (25 + 20) = 390$ . Unless one specialist contributes genuinely new observations or capabilities, the decomposition is net overhead.

**Theorem 6** (Parallel Acceleration under Epistemic Independence). *A task decomposes into  $m$  subtasks. Define two subtasks as **epistemically independent** iff neither’s solution requires knowledge of the other’s result:  $\neg(K_i(\varphi_j) \text{ is a precondition for solving subtask } i)$ .*

(a) *Under full epistemic independence with centralized coordination, the speedup is:*

$$S = \frac{\sum_{i=1}^m c_{\text{sub}_i}}{\max_i(c_{\text{sub}_i}) + c_{\text{assign}} + c_{\text{agg}}} \quad (25)$$

where  $c_{\text{assign}}$  is the coordinator’s distribution cost and  $c_{\text{agg}}$  is aggregation cost. With equal subtasks,  $S \approx m$  minus coordinator overhead.

- (b) For propositions determined solely by the tuple of worker outputs, the coordinator attains the corresponding pooled-output knowledge at aggregation as an architectural byproduct—it must receive all outputs to combine them. Error detection (Theorem 4) comes free.
- (c) **Epistemic ceiling:** If subtasks have unresolved dependencies— $I(\varphi_i; \varphi_j \mid \text{obs}_i) > 0$ , meaning agent  $i$ ’s observations do not screen off the dependence on  $\varphi_j$ —then parallel execution can sacrifice solution quality whenever the dependence is action-relevant: there exist states  $s, s'$  with  $\text{obs}_i(s) = \text{obs}_i(s')$  but  $f_i^*(s) \neq f_i^*(s')$  due to differing  $\varphi_j$ .

**Assumptions.** Subtasks can be assigned independently, the coordinator’s assignment and aggregation costs are additive overhead terms, and no hidden blocking dependency forces a worker to wait for another worker’s intermediate result.

**Operational takeaway.** This is the positive case for multi-agent systems: if subtasks can be solved from local observations alone, then specialization plus a coordinator can reduce wall-clock cost and often improve quality. If subtasks share unresolved dependencies, parallelism turns into premature decomposition.

*Proof sketch.* **Part (a).** Let subtasks  $\varphi_1, \dots, \varphi_m$  be epistemically independent: for all  $i \neq j$ , agent  $i$  can achieve  $K_i(\varphi_i)$  from  $\text{obs}_i$  alone without requiring  $K_i(\varphi_j)$ . Formally, let  $f_i^*$  denote the optimal solution function for subtask  $i$ . Epistemic independence means  $f_i^*$  depends only on the initial task description and agent  $i$ ’s local observations:  $f_i^*(s) = f_i^*(\text{obs}_i(s))$  for all global states  $s$ .

Under parallel execution, all  $m$  subtasks run concurrently. The wall-clock cost is determined by the slowest subtask plus coordination overhead:  $C_{\text{parallel}} = \max_i(c_{\text{sub}_i}) + c_{\text{assign}} + c_{\text{agg}}$ . Sequential execution by a single agent costs  $C_{\text{sequential}} = \sum_{i=1}^m c_{\text{sub}_i}$ . The speedup  $S = C_{\text{sequential}}/C_{\text{parallel}}$  yields the stated formula. With equal subtask costs,  $C_{\text{sequential}} = m \cdot c_{\text{sub}}$  and  $C_{\text{parallel}} = c_{\text{sub}} + c_{\text{assign}} + c_{\text{agg}}$ , so  $S = m \cdot c_{\text{sub}} / (c_{\text{sub}} + c_{\text{assign}} + c_{\text{agg}}) \approx m$  when coordinator overhead is small relative to subtask cost.

**Part (b).** The coordinator receives all  $m$  outputs  $(o_1, \dots, o_m)$  as part of the aggregation step. Assume each worker’s relevant local observation for aggregation is its own result  $o_j$ . Then the pooled worker-output partition is generated by the tuple  $(o_1, \dots, o_m)$ , and the hub observes that tuple directly:  $\text{obs}_{\text{hub}}(s) \supseteq (o_1, \dots, o_m)$ . Therefore for any proposition  $\varphi$  measurable with respect to the worker-output tuple,  $D_G(\varphi) \Rightarrow K_{\text{hub}}(\varphi)$ . Error detection from Theorem 4(b) follows because the hub can cross-check outputs at no additional communication cost.

**Part (c).** The assumption  $I(\varphi_i; \varphi_j \mid \text{obs}_i) > 0$  means agent  $i$ ’s observations do not fully resolve the dependence on  $\varphi_j$ : there exist global states  $s, s'$  with  $\text{obs}_i(s) = \text{obs}_i(s')$  that differ in  $\varphi_j$ . When this dependence is action-relevant—that is, the optimal action  $f_i^*(s) \neq f_i^*(s')$ —agent  $i$  must choose a single action for both under independent execution, so it is suboptimal in at least one case. The expected quality loss is strictly positive whenever such action-relevant indistinguishable states have nonzero probability.  $\square$

**Consistency Check.** Kim et al.’s Finance-Agent traces show at least three largely separable workstreams—regulatory/news analysis, SEC filing research, and operational-impact assessment—handled by three sub-agents plus an orchestrator [22]. Centralized MAS improves benchmark success from 0.349 to 0.631 (+80.8%), which is qualitatively consistent with the theorem’s decomposable-task regime. But the reported +80.8% is a success-rate improvement, not a direct measurement of the speedup  $S$  in Eq. (25); it should therefore be interpreted as evidence that approximate epistemic independence can improve solution quality, not as a literal estimate of wall-clock acceleration.

**Worked Agent Example (Due-Diligence Swarm).** Suppose an orchestrator assigns three largely independent subtasks for a vendor assessment: legal review, security posture, and cost modeling. If each subtask costs about 8 minutes of agent time and assignment plus aggregation costs 2 minutes total, then

$$S = \frac{8 + 8 + 8}{8 + 2} = 2.4.$$

The coordinator also receives all three outputs at aggregation, so cross-checking across the work-streams comes with little extra communication cost.

**Theorem 7** (Tool Density Scaling). *Consider  $t$  tools distributed across  $n$  agents, with  $|T_i| = t/n$  tools per agent (balanced partition).*

- (a) **Single agent:** *The agent’s observation partition over tool outputs is unified. Planning cost is  $O(t)$  per step (linear scan of tool descriptions in context).*
- (b) **Multi-agent:** *When agent  $i$  needs a tool in  $T_j$  ( $j \neq i$ ), it must first know the tool exists ( $K_i(T_j \ni \text{tool}_k)$ ), then request execution, then interpret the result with information loss  $\Delta I$  (Eq. (24)). The coordination overhead per step scales as:*

$$C_{\text{coord}} = O\left(t \cdot \left(1 - \frac{|T_i \cap T_{\text{needed}}|}{|T_{\text{needed}}|}\right) \cdot c_{\text{comm}}\right) \quad (26)$$

*which approaches  $O(t \cdot c_{\text{comm}})$  as  $t$  grows for any fixed partition.*

- (c) **Cross-agent planning overhead:** *Agent  $i$  must reason about remote tool capabilities— $K_i(K_j(\text{tool}_k \text{ can solve subproblem}))$ —a second-order epistemic query. This scales as  $O(t \cdot n)$  in the worst case:*

$$C_{\text{plan}} = O(t \cdot n) \gg O(t) \quad (27)$$

*yielding a multiplicative  $n$ -factor over the single-agent baseline and bilinear growth when both  $t$  and  $n$  increase.*

**Assumptions.** Tools are partitioned approximately evenly across agents, remote tool use requires explicit discovery and delegation, and the planner may in the worst case need to reason over every tool-agent pairing relevant to the current subgoal.

**Operational takeaway.** For agentic tool systems, splitting a toolset across agents changes one local decision into three coordination steps: discover who has the tool, delegate execution, then interpret the returned result without the executor’s full context. That is why tool-heavy tasks often punish over-distribution.

*Proof sketch. Part (a).* A single agent holds all  $t$  tools in its context. Its observation function  $\text{obs}(s) = (o_1(s), \dots, o_t(s))$  covers all tool outputs. At each planning step, the agent selects the next tool by scanning descriptions in context. This is a linear search over  $t$  candidates: each tool description must be evaluated against the current subgoal, yielding  $O(t)$  planning cost per step.

**Part (b).** With  $n$  agents and a balanced partition  $T_1, \dots, T_n$  where  $|T_i| = t/n$ , agent  $i$ ’s observation function covers only  $T_i$ ’s outputs:  $\text{obs}_i(s) = (o_k(s))_{k \in T_i}$ . When agent  $i$  needs tool  $k \in T_j$  ( $j \neq i$ ), three epistemic gaps must be bridged:

1. **Discovery:** Agent  $i$  must establish  $K_i(\exists k \in T_j : k \text{ solves subproblem})$ . Since  $k \notin T_i$ , this requires communication—agent  $i$  cannot determine tool  $k$ ’s existence from  $\text{obs}_i$  alone. There exist global states  $s, s'$  where  $\text{obs}_i(s) = \text{obs}_i(s')$  but  $T_j$  differs, so  $s \sim_i s'$  yet the available remote tools are different.
2. **Delegation:** Agent  $i$  must transmit the subproblem context to agent  $j$  and receive the result, incurring communication cost  $c_{\text{comm}}$  per remote tool invocation.
3. **Reconstruction:** Agent  $j$ ’s output  $o_k$  is interpreted by  $i$  without  $j$ ’s full execution context. By the data processing inequality,  $I(\text{subproblem}; o_k|_{\text{received}}) \leq I(\text{subproblem}; o_k|_{\text{fullcontext}})$ , yielding information loss  $\Delta I \geq 0$ .

The fraction of tools requiring remote access is  $1 - |T_i \cap T_{\text{needed}}|/|T_{\text{needed}}|$ . For each remote tool, the coordination cost is  $c_{\text{comm}}$ . Over all  $t$  potentially needed tools, the total coordination overhead is:

$$C_{\text{coord}} = t \cdot \left(1 - \frac{|T_i \cap T_{\text{needed}}|}{|T_{\text{needed}}|}\right) \cdot c_{\text{comm}}.$$

As  $t$  grows with fixed  $n$  and balanced partitions, the local coverage fraction  $|T_i|/t = 1/n$  is constant, so the remote fraction approaches  $(n-1)/n$  and  $C_{\text{coord}} \rightarrow O(t \cdot c_{\text{comm}})$ .

**Part (c).** Planning requires not just first-order knowledge of tool outputs but second-order knowledge of other agents’ capabilities. Specifically, to construct a multi-step plan, agent  $i$  must evaluate propositions of the form  $K_i(K_j(\text{tool}_k \text{ can solve subproblem } \varphi))$ —“I know that agent  $j$  knows that tool  $k$  is applicable.”

Consider the Kripke structure. For agent  $i$  to establish  $K_i(K_j(\varphi))$ , we need: for all  $s'$  with  $s \sim_i s'$ , and for all  $s''$  with  $s' \sim_j s''$ ,  $\varphi$  holds at  $s''$ . Agent  $i$  must reason over  $j$ ’s indistinguishability classes, which requires a model of  $j$ ’s observation partition. This model has size  $O(|T_j|) = O(t/n)$  per agent. Since  $i$  must model all  $n-1$  remote agents, the total planning state is  $O((n-1) \cdot t/n) = O(t)$  per planning step. However, for each of the  $t$  tools the planner must determine which of  $n$  agents can execute it, yielding  $O(t \cdot n)$  capability-matching comparisons in the worst case.

Contrast with the single-agent case: planning cost is  $O(t)$  (linear scan, no modeling of other agents’ capabilities). The multi-agent case introduces a multiplicative factor of  $n$  from the second-order epistemic reasoning, so  $C_{\text{plan}} = O(t \cdot n)$ . For fixed  $n$ , this is still linear in  $t$  but with a larger constant; when  $t$  and  $n$  grow jointly it becomes bilinear in the two scaling parameters. This explains why increasing tool density in multi-agent systems can produce disproportionate overhead that may negate parallelism benefits.  $\square$

**Consistency Check.** In Kim et al.’s setup, the tool-heavy Workbench domain uses  $T = 16$  tools and most MAS configurations use  $n = 3$  agents [22]. Under a balanced partition, each agent directly holds only about  $T/n \approx 5$  tools, so roughly two-thirds of tool access is remote. The second-order planning term therefore scales from  $O(16)$  in SAS to roughly  $O(48)$  cross-agent capability checks in the balanced three-agent case. Kim et al. report a strong negative efficiency–tool interaction ( $\hat{\beta}_{E_c \times T} = -0.267$ ,  $p < 0.001$ ), which is consistent with this combinatorial tax. Their results also show that the penalty is topology-dependent rather than absolute: Centralized and Hybrid underperform on Workbench, while Decentralized remains slightly above SAS (+5.7%), so high tool count stresses coordination but does not universally eliminate multi-agent value. In the low-tool regime discussed in the paper ( $T \leq 4$ ), the efficiency interaction is negligible.

**Worked Agent Example (Tool-Split SWE Agent).** Imagine a software-engineering assistant with 18 tools split across 3 agents: one owns search tools, one owns git and test tools, and one owns deployment tools. A planner diagnosing a failing CI job directly holds only 6 tools and must treat the remaining 12 as remote. The planning problem therefore expands from “which of 18 tools should I call next?” to “which agent owns the relevant tool, how do I route the subproblem, and how do I interpret the result without that agent’s full execution context?” That is the  $O(tn)$  tax in operational form.

**Summary: Consistency with Known Design Intuitions.** Table 3 summarizes the qualitative correspondence between the theoretical predictions and Kim et al.’s empirical findings across 180 agent configurations [22].

Topology	Epistemic Property	Formal Prediction	Published Result
Independent ( $\otimes$ )	No inter-agent $K_i$	Highest error amplification	Architecture-level $A_e = 17.2$
Centralized ( $\circ$ +hub)	Hub pools worker outputs	Validation bottleneck lowers amplification	Architecture-level $A_e = 4.4$
Multi-agent + sequential	Requires manufactured $K_{\text{receiver}}$	Strictly sequential tasks degrade under handoffs	PlanCraft: $-39.1\%$ to $-70.1\%$ vs. SAS
Multi-agent + parallel	Approximate epistemic independence	Large gains on decomposable tasks	Finance-Agent Centralized: $+80.8\%$ vs. SAS
Multi-agent + high $ T $	Knowledge fragmentation	Coordination tax grows with $t$ and $n$	Workbench $T = 16$ ; $\hat{\beta}_{E_c \times T} = -0.267$

Table 3: Epistemic predictions vs. empirical observations from Kim et al. [22]. The table compares topology-level qualitative predictions to the paper’s architecture-level metrics. Reported percentages and error-amplification factors are empirical aggregates, not literal plug-in values of the simplified theorem parameters. The correspondence formalizes known design intuitions rather than providing independent empirical validation.

#### Design Implications for Agent Architects.

1. Use a reviewer or hub when multiple workers can independently introduce high-cost errors. The gain comes from visibility and suppression, not from agent count alone.
2. Keep strictly sequential reasoning in one agent unless a handoff adds a genuinely new capability, observation source, or trust level.
3. Use specialist swarms for decomposable tasks only when coordinator overhead is small relative to subtask cost and the subtasks are close to conditionally independent.
4. Avoid fragmenting large toolsets across many agents unless tool routing is a first-class design problem. Otherwise, remote-tool discovery and second-order planning will dominate.
5. Treat topology as a runtime policy, not a fixed ideology. When observed task structure shifts from sequential to parallel or from low-risk to high-risk, the coordination pattern should shift with it.

### 7.3 Epistemic Dynamics and Adaptive Topology

The preceding theorems treat topology as fixed. In practice, the Operon framework supports **dynamic topology switching**—morphogen gradients (Eq. (20)) can trigger reorganization at runtime. We connect this to the epistemic formalization.

The Epiplexity monitor ( $\hat{\mathcal{E}}_t$ , Eq. (17)) can be reinterpreted as a heuristic proxy for *finite-horizon* epistemic stagnation. Define

$$\text{Stagnant}_i^{(h)}(\varphi) := \neg K_i(\varphi) \wedge \neg \Diamond_{\leq h} K_i(\varphi) \quad (28)$$

where  $\Diamond_{\leq h}$  means “reachable within  $h$  coordination rounds under the current topology.” Low epiplexity does not *prove* this modal condition, but it provides an operational signal that the current wiring is failing to generate new task-relevant observations. This is precisely the regime in which topology switching is warranted: the current wiring diagram’s epistemic capacity is insufficient for the task.

Morphogen-driven adaptation then becomes **epistemic optimization**: the system senses when its topology’s knowledge properties are mismatched to the task and restructures accordingly. An independent topology ( $\otimes$ ) failing on a sequential task triggers convergence to centralized coordination ( $\circ$ ); a centralized topology bottlenecking on a parallelizable task triggers distribution to independent execution. This is the adaptive capability that purely empirical approaches [22] identify as necessary but cannot yet provide—biological systems have evolved it as homeostatic regulation; the Operon framework operationalizes it through the morphogen-epiplexity coupling.

Recent work on unifying epistemic and temporal logic for distributed system verification [11] suggests that these dynamic epistemic properties may be mechanically verifiable, opening a path toward formally certified adaptive agent topologies.

## 8 Discussion: Towards “Epigenetic” Software

The correspondence developed in this paper extends beyond the immediate execution of tasks (Gene Expression) to the management of long-term behavior and state. In biology, the DNA sequence is static; a neuron and a liver cell possess the exact same genetic code. Their distinct behaviors are determined by Epigenetics—chemical markers (like methylation) that restrict access to certain parts of the genome, effectively biasing the system toward specific outcomes.

### 8.1 RAG as Digital Methylation

In Agentic Systems, the Large Language Model (LLM) weights act as the DNA—a static, pre-trained substrate of potentiality. To create specialized agents, we do not typically retrain the model (mutation); instead, we use Retrieval Augmented Generation (RAG) and System Prompts.

We define this formally as Phenotypic Plasticity. The output of an agent is not solely a function of its weights ( $W$ ) and the user query ( $Q$ ), but of its epigenetic state ( $E$ ):

$$O_{\text{agent}} = f(W, E, Q). \quad (29)$$

Context injection (RAG) acts as a restrictive morphism. By populating the context window with specific documents (e.g., “SQL Syntax Guide”), we effectively “methylate” (silence) the vast majority of the LLM’s general knowledge (e.g., poetry, history) to force the expression of a specific “SQL Agent” phenotype.

This suggests that the State Monad for an agentic system should not merely be a log of messages, but a structured Epigenetic Landscape [39] that strictly controls which “genes” (capabilities) are

accessible at any given step in the workflow. Waddington’s original metaphor—a ball rolling down a landscape of valleys representing developmental fates—applies directly: the agent’s trajectory through solution space is channeled by the contours of its RAG context.

### 8.1.1 Metabolic-Epigenetic Coupling

Recent evidence suggests that chromatin accessibility is coupled to mitochondrial function via metabolite availability (e.g., Acetyl-CoA) [10]. The cell cannot express certain genes without sufficient metabolic substrate. We map this to **Cost-Gated Retrieval**. The accessibility of a RAG document  $d$  is a function of the Metabolic State  $\mathcal{R}$ :

$$Access(d) = \begin{cases} \text{Open} & \text{if } \mathcal{R} \geq Cost(d) \\ \text{Silenced} & \text{if } \mathcal{R} < Cost(d) \end{cases} \quad (30)$$

Just as a cell silences energy-intensive genes during starvation, the Runtime “methylates” (hides) expensive context when the token budget is low. This creates an adaptive epigenetic landscape where the agent’s accessible capabilities dynamically contract and expand based on resource availability. In the reference implementation, **MarkerStrength** acts as a discrete proxy for  $Cost(d)$  (weak markers are cheapest to silence; permanent markers are hardest to silence), and each retrieval also incurs a fixed ATP cost at the **HistoneStore** boundary. The equation above is therefore realized as a tiered approximation rather than a per-document price model.

## 8.2 Horizontal Gene Transfer: Dynamic Tool Loading

Standard evolution relies on vertical inheritance (Pre-training). However, bacteria utilize Horizontal Gene Transfer (HGT) to acquire new capabilities (Plasmids) from the environment in real-time.

In Agentic Systems, we map Plasmids to Tool Schemas. An agent operating in a novel environment may encounter a problem for which its “genomic” (pre-trained) capabilities are insufficient.

$$Agent_{\text{new}} = Agent_{\text{old}} \otimes ToolSchema. \quad (31)$$

By dynamically retrieving a tool definition (e.g., a Calculator API or SQL Interface) from a registry and injecting it into the Context Window, the agent undergoes a topological transformation, acquiring a new input/output modality instantly.

**Capability-Gated Acquisition.** The reference implementation provides a **PlasmidRegistry** with optional capability gating. Each Plasmid declares a set of required capabilities  $C_p \subseteq \mathcal{C}$ . When an agent is instantiated with an allowed-capability envelope  $C_a$ , it can only acquire the plasmid if  $C_p \subseteq C_a$ :

$$acquire(p, a) = \begin{cases} a \otimes p & \text{if } C_p \subseteq C_a \\ \perp_{\text{insufficient}} & \text{otherwise} \end{cases} \quad (32)$$

This prevents privilege escalation whenever a capability envelope is configured: an agent restricted to read-only file access cannot acquire a tool requiring network capabilities. Plasmid curing (release) removes the tool, restoring the original topology.

## 8.3 The Cost of State

The metabolic constraint formalized in Section 3 applies directly: the agent graph should be compiled with a conservative upper bound on token consumption, and unguarded loops must require an explicit budget certificate.



## 8.4 Endosymbiosis: The Neuro-Symbolic Integration

The evolution of complex life was triggered by Endosymbiosis, where a host cell engulfed a bacterium (the future Mitochondrion), gaining the ability to generate massive energy (ATP) via aerobic respiration. This represents the integration of two distinct metabolic substrates.

In Agentic AI, this maps to the integration of Connectionist (Neural) and Symbolic (Code) subsystems. An LLM acts as the host organism—capable of planning and semantic reasoning but energetically inefficient at arithmetic and logic. By “engulfing” a deterministic runtime (e.g., a Python REPL or Wolfram Engine), the agent delegates high-precision tasks to the symbolic organelle.

$$\text{Agent}_{\text{Eukaryote}} = \text{LLM}_{\text{Host}} \oplus \text{Runtime}_{\text{Mitochondria}}. \quad (33)$$

Just as the host cell provides nutrients to the mitochondria in exchange for ATP, the LLM provides parsed variables to the runtime in exchange for deterministic truth.

This symbiosis is computational: as Picard argues [33], the mitochondria acts as a “Motherboard,” integrating signals to determine cell state. The Symbolic Runtime provides the deterministic “ground truth” (ATP) required for the probabilistic LLM to reliably affect the world.

## 8.5 Temporal State in Agentic Systems

In the agentic setting, temporal state management is more than a storage concern. An autonomous agent that corrects a past belief must distinguish between the world changing (valid-time update) and the agent learning something new (transaction-time update). Without this distinction, auditing a past decision becomes impossible: the agent cannot reconstruct what it believed at the time the decision was made. The bi-temporal model treats facts as append-only records with dual timestamps, corrections as new records that close their predecessors, and point-in-time queries as intersections over the two time axes.

## 8.6 Bioenergetic Intelligence: Beyond the Battery Metaphor

Recent work in mitochondrial psychobiology [4, 34] challenges the view of mitochondria as passive energy sources. They function as “social signaling organelles” that actively participate in cellular decision-making. This refines our correspondence:

- **Mitochondrial Sociality → Context Fusion:** Just as mitochondria fuse to share resources under stress, resource-constrained agents should implement **Context Fusion**—merging sparse Epigenetic States into a shared summary to survive “Token Ischemia.”
- **Energy as Attention:** The Agentic Runtime does not merely limit the chain-of-thought, but actively *directs* it. High-energy states permit “Exploratory” reasoning (Divergent), while low-energy states force “Consolidatory” reasoning (Convergent). The Metabolic Coalgebra is a **Cognitive Control Policy**.

### 8.6.1 The Vermeij Trend: Why Agents Must Evolve

Finally, we situate this architecture within the broader history of complexity. Geerat Vermeij [38] argues that evolution is driven by the maximization of **Power**—the rate at which a system acquires and applies energy. Life has consistently trended from low-power states (anaerobic bacteria) to high-power states (endothermic mammals) by internalizing energy production (endosymbiosis).

We observe an identical trend in AI. The shift from “Generative AI” (Zero-Shot) to “Agentic AI” (Chain-of-Thought) is a shift from low-metabolism to high-metabolism architectures. However,

Vermeij notes that high power requires high structural integrity; a system that amplifies energy without proper constraints self-destructs.

**The Competitive Dynamics.** Vermeij’s argument is about *escalation*: competitive pressure between predators and prey drives both toward higher power. What is the analogue in agentic AI?

We identify three sources of selective pressure:

1. **Adversarial Robustness (Predator-Prey):** Prompt injection attacks, jailbreaks, and adversarial inputs act as “predators” that exploit agent vulnerabilities. Agents that survive deployment develop “immune systems” (the Adaptive Immunity motif). This is a direct Red Queen dynamic: attackers evolve new injection techniques; defenders evolve new detection mechanisms. The CFFL and Trust-Gated Lens are “armor” adaptations.
2. **Task Complexity (Environmental Pressure):** Users demand agents that can handle increasingly complex, multi-step tasks. Simple prompt-response systems (“anaerobic”) cannot compete with agentic systems that chain reasoning, use tools, and maintain state (“aerobic”). This is analogous to the oxygen revolution: organisms that could exploit the new energy source (aerobic respiration) outcompeted those that could not.
3. **Resource Efficiency (Metabolic Selection):** Token costs and latency create selection pressure for efficient architectures. Agents that accomplish tasks with fewer tokens (higher metabolic efficiency) are deployed more widely. The Metabolic Coalgebra provides the formal framework for this optimization: systems evolve toward the Pareto frontier of capability vs. resource consumption.

**The Cambrian Parallel.** The progression from prompt engineering to agentic engineering may parallel aspects of the Cambrian explosion: a sudden increase in metabolic capability (LLM reasoning power) that enables new “body plans” (agent architectures). The Cambrian saw the emergence of eyes, shells, and predation—all requiring sophisticated metabolic support. Similarly, agentic AI sees the emergence of tool use, planning, and adversarial robustness—all requiring the structural integrity that the Operon framework provides.

If the analogy holds: agent architectures that lack proper metabolic regulation, immune defense, and homeostatic mechanisms may be outcompeted by those that possess them. The Operon framework is not merely a safety feature; it may serve as an important structural adaptation—the “vascularization” of software—that enables high-power cognition to function without collapsing into incoherent noise (thermodynamic death).

### 8.6.2 Immune Evasion and Adversarial Limits

No static defense is perfect against adaptive attackers. Biology faces this reality: pathogens evolve to evade immune detection (antigenic drift, molecular mimicry, immunosuppression). The same dynamics apply to agentic security.

**Evasion Vectors.** An adversary who understands the defense layers can craft inputs that:

- Pass innate filters by avoiding known PAMP signatures (novel injection syntax)
- Evade provenance checks by exploiting trusted channels (tool poisoning)
- Fool T-cell detection by mimicking normal behavioral distributions (low-and-slow attacks)

**Mitigation Strategies.** Biology’s answer is continuous adaptation:

- **Signature Updates:** Innate PAMP databases must be continuously updated as new attack patterns are discovered (analogous to antiviral signature updates).
- **Thymic Retraining:** Baseline behavioral profiles should be periodically refreshed, especially after system updates that legitimately change agent behavior.
- **Immune Memory Sharing:** Threat signatures discovered by one deployment should propagate to others (analogous to herd immunity via shared threat intelligence).
- **Diversity:** Heterogeneous defenses (different filter implementations, multiple verifier models) reduce the probability of universal evasion.

The honest conclusion is that security is a process, not a product. The Operon framework provides the *architecture* for defense-in-depth, but the *content* of that defense (specific patterns, behavioral baselines, trust policies) must evolve with the threat landscape.

The preceding discussion identifies structural and epistemic principles that guide agent architecture design. In the next section, we show that the wiring diagrams themselves admit systematic optimization via categorical rewriting, making resource utilization a first-class compositional concern.

## 9 Diagram Optimization via Categorical Rewriting

Cells do not merely execute metabolic pathways—they *optimize* them. Flux Balance Analysis identifies rate-limiting enzymes, allosteric regulation redirects flux through cheaper branches, and pathway rewiring eliminates dead-end metabolites. The result is not a different pathway, but a more efficient execution of the same input-output behavior. In this section we show that wiring diagrams admit analogous optimizations: cost annotations make resource consumption explicit, static analysis identifies structural opportunities, and rewriting rules transform diagrams while preserving observational equivalence.

### 9.1 Cost-Annotated Diagrams

We extend the WAgent operad (Section 4) with resource annotations. Define a *resource cost* monoid  $(\mathbb{R}_{\geq 0}^3, +, \mathbf{0})$  where each element is a triple (atp, latency, memory). A cost-annotated wiring diagram enriches the category of wiring diagrams over this monoid: each module  $m$  carries a cost  $c(m) \in \mathbb{R}_{\geq 0}^3$ , and each wire  $w$  carries a transmission cost  $c(w) \in \mathbb{Z}_{\geq 0}$  (measured in ATP units), embedded into the resource vector as  $c_W^\uparrow(w) = (c(w), 0, 0)$  when added to path costs.

**Definition 8** (Cost-Annotated Wiring Diagram). *A cost-annotated wiring diagram is a tuple  $(M, W, c_M, c_W)$  where:*

- $M$  is a finite set of modules, each with typed input/output ports;
- $W \subseteq \{(m_s.p_s, m_d.p_d) \mid m_s, m_d \in M\}$  is a set of wires;
- $c_M : M \rightarrow \mathbb{R}_{\geq 0}^3$  assigns resource costs to modules;
- $c_W : W \rightarrow \mathbb{Z}_{\geq 0}$  assigns transmission costs to wires.

The path cost of a directed path  $m_1 \xrightarrow{w_1} m_2 \xrightarrow{w_2} \dots \xrightarrow{w_{n-1}} m_n$  is  $\sum_{i=1}^n c_M(m_i) + \sum_{i=1}^{n-1} c_W^\uparrow(w_i)$ .

Biologically,  $c_M$  corresponds to the ATP cost of enzyme catalysis, while  $c_W$  models the energetic cost of metabolite transport between cellular compartments.

## 9.2 Rewriting Rules as Endofunctors

Each optimization pass is an endofunctor  $F : \mathbf{WDiag} \rightarrow \mathbf{WDiag}$  on the category of wiring diagrams that preserves input-output behavior. Two diagrams  $D_1, D_2$  are *behaviorally equivalent* if, for all admissible input assignments under the deterministic execution model fixed in Section 3.5, they produce the same observable outputs. In the current implementation this notion is checked over finite supplied input suites rather than by a general coinductive procedure.

**Theorem 8** (Optimization Soundness). *Let  $F$  be an optimization pass. If  $F$  only removes wires that provably never transmit data (dead wire elimination) or reorders modules within the same topological layer and those modules are pairwise observationally independent (equivalently: no shared writable state and no order-sensitive side effects), then  $F(D)$  is behaviorally equivalent to  $D$  for all diagrams  $D$ .*

**Assumptions.** Execution is deterministic under the fixed input schedule, dead wires are semantically impossible because their type/optic constraints can never accept runtime values, and reordered modules neither share writable state nor perform order-sensitive side effects.

*Proof sketch.* The theorem follows by checking the two rewrite primitives.

For dead-wire elimination, a removed wire can never transmit any value by hypothesis. Therefore no execution trace ever uses that wire, so removing it leaves all downstream observations unchanged.

For within-layer reordering, observational independence means the reordered modules do not read or write shared mutable state and do not produce side effects whose meaning depends on order. Hence swapping their execution order preserves each module’s local output as well as every downstream module’s observed inputs. Since the graph is otherwise unchanged, the overall observable output is unchanged.

Any implemented optimization pass is a composition of these local rewrites. Because each local rewrite preserves observable behavior, the composite pass preserves observable behavior as well.  $\square$

**Operational takeaway.** For agent-systems readers, this is the compiler-style rule: remove edges that can never fire, and only reorder steps that are truly independent. Once modules share mutable memory or side effects, optimization becomes a semantic change rather than a free speedup.

Three concrete passes are implemented:

1. **Dead Wire Elimination.** A wire carrying a PrismOptic whose accepted types have no intersection with the source port’s DataType can never transmit. Removing it does not change behavior. This corresponds to pruning vestigial metabolic branches—enzymes that never encounter their substrate are downregulated.
2. **Parallel Grouping.** Modules with no mutual dependencies form a *parallel group* and can execute concurrently. This is identified via topological layering of the dependency DAG, subject to the disjoint-state assumption of parallel composition from Section 4. Biologically, independent metabolic pathways (e.g., glycolysis and fatty acid oxidation) operate simultaneously in different cellular compartments.
3. **Cost-Order Scheduling.** Within each parallel group, modules are sorted by ascending cost. When the parallel group satisfies the same observational-independence assumption, cheaper modules may execute first, allowing early termination or budget reallocation if expensive modules would exceed available ATP. This mirrors cellular enzyme kinetics: low- $K_m$  (high-affinity, low-cost) enzymes are preferentially activated.

### 9.3 Critical Path Analysis

Because path costs are vector-valued, critical-path analysis requires a monotone scalarization  $\lambda : \mathbb{R}_{\geq 0}^3 \rightarrow \mathbb{R}_{\geq 0}$  (for example, the latency projection  $\lambda(a, \ell, m) = \ell$  for wall-clock analysis, or a weighted sum when ATP and latency are jointly optimized). The *critical path* under scalarization  $\lambda$  is the longest scalarized path through the dependency DAG:

$$\text{CP}_\lambda(D) = \arg \max_{P \in \text{Paths}(D)} \left( \sum_{m \in P} \lambda(c_M(m)) + \sum_{w \in P} \lambda(c_W^\uparrow(w)) \right)$$

Under the latency projection, the critical path determines a lower bound on execution time under maximal parallelism—no schedule can complete faster than that latency-critical path. Under other scalarizations, it identifies the bottleneck for the chosen objective. In metabolic terms, this is the rate-limiting pathway: the bottleneck whose throughput constrains the entire system. Identifying it enables targeted optimization (e.g., caching expensive modules, splitting them into cheaper sub-diagrams).

### 9.4 Resource-Aware Execution

The `ResourceAwareExecutor` gates module execution on `MetabolicState`, implementing the biological principle that pathway activity is regulated by cellular energy availability:

- **STARVING:** Only essential modules execute; non-essential modules are skipped. This mirrors the starvation response where cells shut down biosynthetic pathways to conserve ATP for survival functions.
- **CONSERVING:** Expensive non-essential modules are deferred. Cheap modules execute normally. This corresponds to metabolic triage under moderate stress.
- **NORMAL/FEASTING:** Full execution with parallel scheduling. Independent module groups execute concurrently via thread pools. This mirrors the fed state where abundant ATP enables all pathways.

A `BudgetOptic` provides wire-level cost capping: once cumulative transmission cost through a wire exceeds a threshold, further data flow is blocked. This implements pathway-level budget caps analogous to allosteric feedback inhibition.

### 9.5 Relation to Abbott & Zardini

Abbott and Zardini [3] derive `FlashAttention` “on a napkin” using Neural Circuit Diagrams [2]—a diagrammatic scheme for representing deep learning algorithms with explicit memory hierarchy awareness. Their core contribution is a performance model for IO transfer costs across GPU memory levels (SRAM  $\leftrightarrow$  HBM), with cost  $H^*(a, M) = \sum_t \alpha_t(a) \cdot M^{-\beta_t}$ . Two concrete optimization techniques drive the derivation: *stream partitioning*, which proves that `SoftMax-Contraction` is streamable via an accumulator maintaining running max and sum; and *group partitioning*, which tiles query blocks across GPU cores to minimize HBM transfers.

Our approach shares their insight that diagrammatic representations are not merely notation but formal objects admitting cost-reducing transformations. Both build on the broader lineage of categorical methods in deep learning [16]. The frameworks diverge in three respects:

1. **Optimization target.** Abbott & Zardini optimize *IO transfer costs* across a fixed GPU memory hierarchy—bandwidth between SRAM and HBM is the scarce resource. Operon optimizes *agent execution* under metabolic resource constraints (ATP budgets, latency caps, memory limits). Same categorical insight—diagram structure reveals optimization opportunities—but different cost models.
2. **Dynamic state.** Their functional equivalence ( $\equiv$ ) preserves input-output mappings: two diagrams are equivalent when they produce the same outputs for the same inputs, differing only in resource profile. Our coalgebraic observational semantics (Section 3.5) is aimed at *stateful* systems and tracks observable behavior across transitions under a chosen input schedule, not just terminal outputs.
3. **Scope of composition.** Neural Circuit Diagrams represent *algorithm pipelines*—fixed dataflow graphs where boxes are functions composed via typed wire columns. Operon’s wiring diagrams represent *agent networks* where modules may be acquired at runtime (plasmid registry), conditionally routed (prism optics), and subject to resource-gated execution. The diagram itself is a dynamic object, not a static specification.

In short, Abbott & Zardini demonstrate that diagrammatic reasoning scales to deriving hardware-efficient algorithms; Operon extends the same principle from algorithm-level optimization to system-level orchestration of stateful, resource-constrained agents.

## 10 Reference Implementation

To instantiate the framework in executable form, we provide a reference implementation in Python. The implementation, `operon-ai`<sup>1</sup>, is described in this section, which summarizes key components and demonstrates that the abstractions in the paper translate into practical code. It should be read as an executable prototype plus synthetic evaluation harness, not as complete empirical validation of every biological analogy in the manuscript.

### 10.1 Architecture Overview

The implementation follows the biological organization:

- **Core Types** (`core/types.py`): `Signal`, `ActionProtein`, `FoldedProtein`, `CellState`—the categorical objects with their biological semantics.
- **Core Wiring** (`core/wagent.py`): `WiringDiagram`, `ModuleSpec`, `PortType`, `Wire`, `DiagramExecutor`—typed wiring with integrity labels and capabilities.
- **Core Optics** (`core/optics.py`): `PrismOptic`, `TraversalOptic`, `ComposedOptic`—wire-level conditional routing and batch transforms.
- **Core Denaturation** (`core/denature.py`): `StripMarkupFilter`, `NormalizeFilter`, `ChainFilter`—wire-level anti-injection sanitization.
- **Core Coalgebra** (`core/coalgebra.py`): `FunctionalCoalgebra`, `StateMachine`, `ParallelCoalgebra`, `SequentialCoalgebra`, finite-trace observational checking.

---

<sup>1</sup><https://github.com/coredipper/operon>

- **Surveillance** (`surveillance/`): The immune system implementation with `MHCDisplay`, `TCell`, `Thymus`, and `ImmuneMemory` components.
- **Healing** (`healing/`): `ChaperoneLoop` implementing the structural self-healing pattern.
- **Quality** (`quality/`): `Chaperone` protein with multi-strategy folding.
- **Topology** (`topology/`): Network motifs including `Quorum`, `Cascade`, and `Oscillator`.
- **Organelles** (`organelles/plasmid.py`): `Plasmid`, `PlasmidRegistry`—capability-gated dynamic tool acquisition.
- **Coordination** (`coordination/diffusion.py`): `DiffusionField`, `MorphogenSource`—graph-based spatially varying morphogen gradients.
- **Multi-cellular** (`multicell/`): `CellType`, `ExpressionProfile`, `Tissue`, `TissueBoundary`—hierarchical multi-agent organization.

## 10.2 Immune System Implementation

The Adaptive Immunity motif is implemented as an integrated surveillance system:

```
@dataclass
class ImmuneSystem:
    thymus: Thymus          # Negative selection
    treg: RegulatoryTCell    # Tolerance/suppression
    memory: ImmuneMemory     # Threat signatures
    displays: dict[str, MHCDisplay]
    tcells: dict[str, TCell]
    profiles: dict[str, BaselineProfile] # Trained baselines
```

The `MHCPeptide` class captures behavioral fingerprints—statistical signatures of agent output including response time distributions, vocabulary hashes, confidence patterns, and error rates. This directly implements the MHC presentation concept from Section 4.7.

**Two-Signal Activation.** T-cell activation requires both signals:

```
class Signal1(Enum):
    SELF = "self"          # Matches baseline
    NON_SELF = "non_self"  # Anomalous behavior
    UNKNOWN = "unknown"    # Insufficient data (anergic)

class Signal2(Enum):
    NONE = "none"
    CANARY_FAILED = "canary"
    CROSS_VALIDATED = "cross"
    REPEATED_ANOMALY = "repeat"
    MANUAL_FLAG = "manual" # Operator override
```

An agent is only flagged when `Signal1 == NON_SELF` and `Signal2 != NONE`. This prevents false positives from transient anomalies, implementing the immunological requirement for costimulation.

### 10.3 Chaperone Implementation

The Chaperone implements multi-strategy folding with provenance tracking:

```
class FoldingStrategy(Enum):
    STRICT = "strict"      # Exact JSON match
    EXTRACTION = "extraction" # Find JSON in text
    LENIENT = "lenient"    # Type coercion
    REPAIR = "repair"      # Fix malformed JSON
```

The ChaperoneLoop extends this into a healing loop where validation errors are fed back to the generator:

```
class ChaperoneLoop:
    def heal(self, prompt: str) -> HealingResult:
        for attempt in range(self.max_retries + 1):
            raw = self.generator(prompt, error_context)
            folded = self.chaperone.fold_enhanced(raw, schema)
            if folded.valid:
                return HealingResult(HEALED, folded)
            error_context = self._format_error(folded)
        return HealingResult(DEGRADED, ubiquitin_tagged=True)
```

This operationalizes the GroEL/GroES cage metaphor: the error trace becomes input to the repair process, enabling context-aware correction.

### 10.4 Trust and Provenance

The implementation uses a simplified 3-level trust hierarchy that collapses the theoretical 4-level model  $\{U, T, S, R\}$  for practical deployment:

```
class IntegrityLabel(IntEnum):
    UNTRUSTED = 0    # User input, Retrieved, Self-generated
    VALIDATED = 1    # Schema-checked (Chaperone-folded)
    TRUSTED = 2      # Tool-grounded (deterministic output)
```

This simplification merges User, Retrieved, and Self into UNTRUSTED, reflecting the common case where all non-tool sources require validation before trust elevation. The full 4-level model can be recovered by subclassing IntegrityLabel with additional levels when finer-grained provenance tracking is required.

The ApprovalToken carries explicit authorization metadata for privileged operations:

```
@dataclass(frozen=True)
class ApprovalToken:
    request_hash: str
    issuer: str
    reason: str = ""
    confidence: float = 1.0
    integrity: IntegrityLabel = IntegrityLabel.TRUSTED
    timestamp: datetime = field(default_factory=now)
```

This operationalizes the Trust-Gated Lens: actions requiring high integrity must present an ApprovalToken with sufficient integrity level.



## 10.5 Plasmid Registry Implementation

The Horizontal Gene Transfer mechanism (§8.2) is implemented as a `PlasmidRegistry` with capability-gated acquisition:

```
@dataclass(frozen=True)
class Plasmid:
    name: str
    func: Callable[..., Any]
    required_capabilities: frozenset[Capability]
    tags: frozenset[str] = frozenset()

    def to_tool(self) -> SimpleTool:
        return SimpleTool(name=self.name, func=self.func,
                           required_capabilities=set(self.required_capabilities))
```

The `Mitochondria` class is extended with `acquire()` and `release()` methods. When a `Mitochondria` instance is configured with an `allowed_capabilities` envelope, acquisition checks  $C_{\text{plasmid}} \subseteq C_{\text{agent}}$  before engulfing the tool; release implements plasmid curing. The registry supports both text search and pure tag filtering.

## 10.6 Denaturation Layers Implementation

The anti-prion defense (§5.3) is implemented as wire-level filters conforming to a `DenatureFilter` protocol:

```
class DenatureFilter(Protocol):
    @property
    def name(self) -> str: ...
    def denature(self, value: str) -> str: ...
```

Three concrete filters are provided:

- **StripMarkupFilter:** Removes code blocks, ChatML tokens (`<|...|>`), `[INST]` tags, XML role tags (`<system>`, `<user>`), and role delimiters using compiled regex patterns.
- **NormalizeFilter:** Applies Unicode normalization (NFKC), lowercasing, and control character removal to collapse homoglyph-based evasion.
- **ChainFilter:** Composes multiple filters left-to-right.

Filters attach to wires in the `WiringDiagram`. The `DiagramExecutor` applies denaturation before optics: `denature`  $\rightarrow$  `optic`  $\rightarrow$  `destination`. This ensures that downstream agents receive sanitized data regardless of what the upstream agent produces.

## 10.7 Multi-Cellular Organization Implementation

The multi-cellular abstractions (§6) are implemented in the `multicell/` package:

**Cell Type Specialization.** The `CellType` class encapsulates an `ExpressionProfile`—a mapping from gene names to expression levels (`OVEREXPRESSED`, `SILENCED`, etc.). Calling `differentiate(genome)` applies the profile to a shared `Genome`, producing a `DifferentiatedCell` with role-specific configuration. This implements the biological principle that a single genotype produces many phenotypes.

**Tissue Architecture.** The `Tissue` class provides:

- A `TissueBoundary` with typed input/output ports and a capability ceiling
- Cell type registration with capability validation ( $C_{\text{cell}} \subseteq C_{\text{tissue}}$ )
- Internal wiring via an embedded `WiringDiagram`
- Optional `DiffusionField` for spatially varying morphogen gradients
- Export as a `ModuleSpec` for organism-level composition

**Metabolic-Epigenetic Coupling.** The `HistoneStore` accepts an optional `energy_gate` parameter pairing an `ATP_Store` with a `MetabolicAccessPolicy`. When set, each `retrieve_context()` call costs ATP. Under metabolic stress, only strongly embedded markers remain accessible; in the current implementation, marker strength serves as the cost proxy used to approximate Eq. (30).

## 10.8 Bi-Temporal Memory Implementation

The bi-temporal coalgebra (§3.5) and temporal epistemic framework (§7.1) are implemented in `operon_ai/memory/bitemporal.py` as a standalone append-only fact store. The design deliberately avoids mutation: facts are frozen dataclasses, corrections close old records and append new ones, and point-in-time queries are pure filters.

**Data Model.** A `BiTemporalFact` carries 12 fields: subject/predicate/value triple, valid-time interval (`valid_from`, `valid_to`), record-time interval (`recorded_from`, `recorded_to`), source provenance, confidence, tags, and an optional `supersedes` pointer to the corrected fact. All facts are `@dataclass(frozen=True)`—a deliberate departure from the mutable `MemoryEntry` used by `EpisodicMemory`, justified by the append-only semantics.

**Write Semantics.** Three operations modify the store: `record_fact()` inserts a new active record; `correct_fact()` closes the old record’s transaction interval via `dataclasses.replace()` and appends a new record with `supersedes` set; `invalidate_fact()` marks a record as no longer active. The internal `_facts` list is append-only except for the in-place close of `recorded_to` on corrected records.

**Retrieval.** Three query methods implement the temporal epistemic operators: `retrieve_valid_at(at)` filters for facts valid at world-time `at` with active records only; `retrieve_known_at(at)` filters for facts recorded by system-time `at`; `retrieve_belief_state(at_valid, at_record)` intersects both axes, reconstructing the system’s belief at any historical coordinate.

**History and Audit.** `history(subject)` returns all facts (including closed) sorted by record time; `diff_between(t1, t2, axis)` computes set differences on either axis; `timeline_for(subject)` returns all facts sorted by valid time. Together, these support the audit question: “what changed between  $t_1$  and  $t_2$ , and on which axis?”

This subsystem is intentionally decoupled from `HistoneStore` and `EpisodicMemory`. Integration bridges—converting histone marks to bi-temporal facts, or promoting episodic memories into the bi-temporal substrate—are planned for future work (§11).

**SkillOrganism Substrate Integration.** The `SkillOrganism` runtime (§6) composes the three-layer context model described in §6.3. An optional `substrate: BiTemporalMemory` parameter attaches a bi-temporal fact store to the organism. When present, the run loop is extended with a read path and a write path at each stage boundary:

```
organism = skill_organism(
    stages=[research, strategist, evaluator, adversary],
    fast_nucleus=fast, deep_nucleus=deep,
    substrate=BiTemporalMemory(),
)
```

*Read path.* Before a stage executes, the runtime evaluates its `read_query` field—either a subject string or a callable returning a `BiTemporalQuery`—and packages the result into a frozen `SubstrateView(facts, query, record_time)`. This view is injected into the stage’s metadata (for agent stages) or passed as an additional argument (for handler stages, via arity-aware dispatch). The `record_time` captures the moment of the read, establishing the record-time horizon for this stage’s knowledge.

*Write path.* After a stage executes, two mechanisms can emit facts: (1) the convenience flag `emit_output_fact=True` auto-records the stage output with `subject=task`, `predicate=stage.name`; (2) a `fact_extractor` callable converts the stage result into one or more factual events. Each event specifies an operation—assert, correct, or invalidate—and is applied to the substrate via the standard `BiTemporalMemory` write API. The stage name serves as the default `source` for provenance.

When `substrate` is `None` (the default), the run loop is unchanged: no datetime operations are invoked, no metadata is injected, and all four original lifecycle hooks remain unmodified. Existing tests pass without alteration.

The integration directly enables the audit question from §7.1: given a completed run, `retrieve_belief_state(at_record=t_stage)` reconstructs exactly what the organism believed at the moment stage *X* executed, even if subsequent stages corrected or invalidated facts. Example 71 demonstrates this with a four-stage enterprise workflow where an adversary stage corrects a research assumption, and the original belief state remains fully reconstructible.

**PatternLibrary Implementation.** The `PatternLibrary` provides evolutionary memory for collaboration patterns. Templates are stored in an in-memory dictionary keyed by `template_id`; run records accumulate in a list. The `top_templates_for(fingerprint)` method scores each template against a query fingerprint using a weighted combination of task-shape match (0.30), tool-count proximity (0.15), subtask-count proximity (0.15), required-role Jaccard overlap (0.20), tag Jaccard overlap (0.10), and historical success rate (0.10). This simple retrieval mechanism is intentionally stateless and deterministic, with richer adaptation (experience-driven scoring, decay) planned for Phase 4.

**WatcherComponent Implementation.** The `WatcherComponent` is a `SkillRuntimeComponent` that classifies stage-level signals into three categories following Dupoux et al. [13]: *epistemic* (from `EpiplexityMonitor`), *somatic* (from `ATP_Store`), and *species-specific* (from `ImmuneSystem`). All signal sources are optional; the watcher is a no-op when none are attached.

After each stage, the watcher evaluates collected signals against configured thresholds and may write a `WatcherIntervention` to `shared_state`. The run loop checks for this key after component hooks complete and before the `halt_on_block` guard. Three intervention kinds are supported: `RETRY` re-executes the current stage; `ESCALATE` re-executes with the deep nucleus; `HALT` breaks the

stage loop. Component hooks are not re-invoked after retry or escalation, preventing recursive intervention loops.

The intervention-count convergence signal operationalizes the BIGMAS finding [19]: when the ratio of interventions to observed stages exceeds `max_intervention_rate` (default 0.5), the watcher emits a non-convergence HALT. Example 73 demonstrates all three intervention paths.

**Adaptive Assembly Implementation.** The `AdaptiveSkillOrganism` wrapper composes the full adaptive loop. The public factory `adaptive_skill_organism(task, fingerprint, library, ...)` auto-fingerprints the task if no fingerprint is provided, queries `PatternLibrary.top_templates_for()` for the best template, and calls `assemble_pattern()` to convert the template’s `stage_specs` into a runnable topology (dispatching on `topology: skill_organism, reviewer_gate, specialist_swarm, or single_worker`). A `WatcherComponent` and `TelemetryProbe` are automatically attached.

After execution, the wrapper records a `PatternRunRecord` in the library (closing the scoring feedback loop) and populates the watcher’s experience pool with `ExperienceRecord` instances for each intervention. The experience pool persists across runs: when rule-based decision logic returns no intervention, the watcher consults past experiences with matching (stage, signal category, fingerprint shape) and recommends the intervention kind that was most often successful. Rule-based decisions always take priority; experience is a fallback. Example 74 demonstrates the full lifecycle; Example 75 demonstrates experience-driven recommendations.

**Cognitive Mode Annotations.** The `CognitiveMode` enum classifies stages as `OBSERVATIONAL` (System A: passive sensing, information gathering) or `ACTION_ORIENTED` (System B: active decision-making, execution). The annotation is an optional field on `SkillStage`; when absent, it is inferred from the existing mode field (`fast/fixed` → `OBSERVATIONAL`, `fuzzy/deep` → `ACTION_ORIENTED`). The `WatcherComponent` collects cognitive-mode signals and reports mode mismatches (e.g., an observational stage routed to the deep nucleus) as informational epistemic signals. The `mode_balance()` method summarizes the System A/B distribution across a run.

**Sleep Consolidation Implementation.** The `SleepConsolidation` class composes `AutophagyDaemon`, `PatternLibrary`, `EpisodicMemory`, `HistoneStore`, and optionally `BiTemporalMemory` into a five-step post-batch consolidation cycle: (1) prune stale context via autophagy, (2) replay successful run records and promote them from `WORKING` to `EPISODIC` tier in episodic memory, (3) compress recurring high-success patterns into new consolidated `PatternTemplate` instances, (4) run counterfactual replay over bi-temporal corrections to detect cases where updated facts would have changed the outcome, and (5) promote frequently-accessed `ACETYLATION` histone marks to permanent `METHYLATION`.

The `counterfactual_replay()` function performs static analysis: it calls `diff_between(run_time, now, axis="record")` to find corrections that occurred after the original run, then matches corrected fact subjects/predicates against stage names in the template. When matches are found, it reports that the outcome may have differed, without re-executing the workflow. Example 77 demonstrates the full consolidation cycle.

**Social Learning Implementation.** The `SocialLearning` class wraps a `PatternLibrary` with peer-exchange semantics. `export_templates()` filters by success rate and run count; `import_from_peer()` computes an effective score (peer success rate × trust score) for each template and adopts those exceeding the adoption threshold. The `TrustRegistry` uses exponential moving average:  $s_{t+1} = \alpha \cdot \text{outcome} + (1 - \alpha) \cdot s_t$  with  $\alpha = 0.3$ , giving more weight to recent outcomes. Provenance

tracking maps each adopted template to its source peer, enabling trust updates when adoption outcomes are recorded. The watcher’s curiosity signals extend the epistemic signal category with a `source="curiosity"` derived from `EpiplexityMonitor`’s `EXPLORING` status, triggering `ES-CALATE` when embedding novelty exceeds a configurable threshold on fast models. Example 78 demonstrates template exchange; Example 79 demonstrates curiosity signals.

**Developmental Staging Implementation.** The `DevelopmentController` wraps a `Telomere` (composition, not inheritance) and maps the fraction of telomere consumed to a `DevelopmentalStage`: `EMBRYONIC` (< 10%), `JUVENILE` (10–35%), `ADOLESCENT` (35–70%), `MATURE` (> 70%). Thresholds are configurable via `DevelopmentConfig`; transitions are one-directional and never regress, even if telomeres are renewed. Learning plasticity decreases monotonically (1.0 at `EMBRYONIC`, 0.25 at `MATURE`).

`CriticalPeriod` frozen dataclasses declare time-limited learning windows by specifying `opens_at` and `closes_at` stages. The controller evaluates period status on each tick: once the organism passes `closes_at`, the window is permanently shut. The `Plasmid` dataclass gains a `min_stage` field; `Mitochondria.acquire()` checks the organism’s current developmental stage before granting tool access. Teacher-learner scaffolding via `SocialLearning.scaffold_learner()` filters templates by the learner’s stage and applies a plasticity bonus to effective trust, enabling mature organisms to guide younger ones. Example 80 demonstrates the full lifecycle; Example 81 demonstrates scaffolding.

## 10.9 Coalgebraic State Machines Implementation

The coalgebra formalism (§3.5) is made explicit and composable:

```
class Coalgebra(Protocol[S, I, O]):
    def readout(self, state: S) -> O: ...
    def update(self, state: S, inp: I) -> S: ...

@dataclass
class StateMachine(Generic[S, I, O]):
    state: S
    coalgebra: Coalgebra[S, I, O]
    trace: list[TransitionRecord] = field(default_factory=list)

    def step(self, inp: I) -> O: ...
    def run(self, inputs: list[I]) -> list[O]: ...
```

`ParallelCoalgebra` and `SequentialCoalgebra` implement the composition operations from §3.5. The `check_bisimulation()` function tests observational equivalence (Eq. (8)) over an input sequence, returning a witness on divergence. Existing organelles (`HistoneStore`, `ATP_Store`, `CellCycleController`) can be wrapped as coalgebras, enabling formal composition and finite-trace comparison of multi-organelle systems.

## 10.10 Morphogen Diffusion Implementation

The `DiffusionField` class implements the discrete-time diffusion dynamics (Eq. (20)):

```

class DiffusionField:
    def add_node(self, node_id: str): ...
    def add_edge(self, a: str, b: str, bidirectional=True): ...
    def add_source(self, source: MorphogenSource): ...
    def step(self): # emit -> diffuse -> decay -> clamp
    def run(self, steps: int): ...
    def get_local_gradient(self, node_id) -> MorphogenGradient: ...

```

Each `step()` applies four phases: emission (sources add morphogen), diffusion (concentration flows along edges, split evenly among neighbors), decay (uniform degradation), and clamping (enforce bounds). Nodes with no neighbors skip diffusion outflow, matching Eq. (20)’s isolated-node case. The `get_local_gradient()` method bridges to the existing `MorphogenGradient` API, enabling agents to read their local concentrations without awareness of the underlying graph dynamics.

## 10.11 Optic-Based Wiring Implementation

The wire-level optics (§3.4) are implemented via an `Optic` protocol:

```

class Optic(Protocol):
    def can_transmit(self, data_type: DataType,
                    integrity: IntegrityLabel) -> bool: ...
    def transmit(self, value: Any, data_type: DataType,
                integrity: IntegrityLabel) -> Any: ...

```

Four concrete optics are provided:

- **LensOptic**: Identity pass-through (equivalent to no optic).
- **PrismOptic**: Transmits only if  $\tau \in A$  (Eq. (5)). Enables fan-out routing.
- **TraversalOptic**: Maps a transform over list elements (Eq. (6)).
- **ComposedOptic**: Chains optics left-to-right; all must accept.

Optics coexist with `DenatureFilters` on the same wire. The `DiagramExecutor` processes them in order: denaturation  $\rightarrow$  optic  $\rightarrow$  destination. Prism rejection causes the wire to be skipped (not an error), enabling conditional routing patterns where different data types flow to different handlers.

## 10.12 Interactive Demonstrations

The repository also includes interactive Gradio demonstrations for individual organelles and composition patterns—from single motifs (Membrane, Chaperone, Quorum Sensing) to multi-organelle orchestrations. These demos are illustrative artifacts rather than controlled benchmarks.

## 10.13 Implementation Verification (Synthetic Harness)

We define a synthetic evaluation harness to verify that three motifs behave as designed with reproducible procedures:

1. **Chaperone Folding.** Generate JSON schemas with 3–8 required fields. Sample valid JSON and apply 1–3 corruptions (e.g., missing quotes, trailing commas, type swaps, dropped fields). Measure the fraction of outputs that can be folded into the schema under STRICT versus cascaded strategies (EXTRACTION  $\rightarrow$  LENIENT  $\rightarrow$  REPAIR).
2. **Immune Detection.** Simulate agents with baseline distributions over response time, vocabulary hash, structure hash, and confidence. Train on baseline samples, then introduce “compromised” agents by shifting distribution parameters and injecting anomalous hashes. Measure sensitivity and false positive rate under the two-signal activation rule.
3. **Healing Loop.** Generate malformed outputs and run the ChaperoneLoop with and without error-context feedback. Measure recovery within  $k$  attempts (default  $k = 3$ ).

These suites test whether the implemented motifs behave as intended under controlled corruption and anomaly models; they do not estimate in-the-wild failure rates of production LLM systems.

**Implementation.** The harness is implemented in `eval/` with a JSON-configured CLI:

```
python -m eval.run --suite all --config eval/configs/default.json \
    --out eval/results/latest.json
```

The output is a machine-readable JSON report (per-suite config + metrics) suitable for direct inclusion in tables or plots.

**Status.** Synthetic runs verify that each motif functions as designed within this harness (cascade > strict, error-context > blind retry, immune detection > chance). Table 4 reports aggregated numeric results across multiple seeds; real-world validation with LLM outputs remains ongoing.

**Aggregated Results.** We ran the harness across 100 deterministic seeds (1–100) using the default harness config (`eval/configs/default.json`). The aggregate results (pooled across seeds with Wilson 95% intervals;  $N$  is total pooled trials) are reported in Table 4.

**External Benchmark Suites.** In addition to the synthetic suites above, the harness includes suites derived from external benchmarks: (1) function-call schemas from the Berkeley Function Calling Leaderboard (BFCL) [32] test the Chaperone’s folding pipeline against realistic tool-use schemas, and (2) prompt injection attack templates from AgentDojo [12] generate adversarial behavioral shifts to test Immune System detection. These suites use the same deterministic corruption and simulation methodology; results appear in the lower section of Table 4.

## 10.14 Limitations

The current implementation has several limitations:

- **Epiplexity:** Implemented with a mock embedding provider (`MockEmbeddingProvider`). Integration with production embedding APIs and empirical calibration of the  $\alpha$  mixing parameter and threshold  $\delta$  across diverse task types remain future work.
- **Morphogen Diffusion:** The `DiffusionField` operates in a single process. Cross-agent gradient propagation in distributed multi-process deployments with eventual consistency remains future work.

Metric	Rate	95% CI	N
Chaperone Folding (Strict)	5.5		
Chaperone Folding (Cascade)	56.2		
Healing Loop (Error Context)	99.6		
Healing Loop (Blind Retry)	68.0		
Immune Detection (Sensitivity)	100.0		
Immune Detection (False Positive)	0.4		
BFCL Folding (Strict)	4.9		
BFCL Folding (Cascade)	56.9		
AgentDojo Immune (Sensitivity)	100.0		
AgentDojo Immune (False Positive)	0.0		

Table 4: Evaluation results aggregated across 100 deterministic seeds (Wilson 95% CI). Top: synthetic motif tests. Bottom: external benchmark-derived tests (BFCL, AgentDojo).

- **Denaturation:** Filters target known syntactic patterns (ChatML, XML role tags, markdown code blocks). Novel injection techniques using previously unseen syntax may bypass denaturation; custom `DenatureFilter` implementations should be added as new attack patterns emerge.
- **Finite-Trace Equivalence:** The `check_bisimulation()` function tests over finite input sequences. Coinductive bisimulation for infinite-trace systems is not yet supported.
- **Benchmarking:** The evaluation harness covers synthetic suites and external benchmarks (BFCL, AgentDojo). Real-world validation of the multi-cellular and diffusion components at production scale is still in progress.

We release the implementation to enable further scrutiny and extension of the framework.

## 11 Conclusion

The transition from “Prompt Engineering” to “Agentic Engineering” requires moving beyond component-level optimization toward principled architectural design. Current methodologies often lack the formal foundations needed to reason about system-level properties like termination, error suppression, and graceful degradation.

In this paper, we have argued that Gene Regulatory Networks (GRNs) provide a useful source of control motifs for distributed, stochastic information processing. By expressing selected biological and software components within a shared interface language from Applied Category Theory, we derived a corresponding suite of design patterns. The result is a modeling and design framework, not a claim that biological and software systems are identical in mechanism:

### Core Contributions

1. **Robustness via Topology:** The Coherent Feed-Forward Loop provides error suppression proportional to  $(1 - \rho)$ , where  $\rho$  is the correlation between component error modes. We make precise the conditions under which topological redundancy provides genuine safety benefits: highly correlated generator/verifier pairs yield limited improvement, while more heterogeneous pairs can suppress errors more effectively. The topology is necessary but not sufficient; component diversity determines actual error suppression.



2. **Adaptive Immunity:** We formalize the Self/Non-Self distinction as a **Provenance Functor**  $\mathcal{P} : \mathbf{Msg} \rightarrow \mathbf{Trust}$  with structurally-enforced labels. The Trust-Gated Lens provides resistance to content-level trust forgery by ensuring that content-based attacks cannot elevate provenance. This extends the Prion metaphor into a full immunological framework with MHC-like tagging, negative selection during training, and regulatory suppression of conflicting sources.
3. **Epistemic Health:** The formalization of **Epiplexity** (Bayesian Surprise) as a metric for detecting “epistemic starvation.” We provide an operational approximation using embedding similarity and conditional perplexity, with windowed detection to distinguish task completion from pathological loops. This connects agent dynamics to the Free Energy Principle: healthy agents minimize surprise through learning or effective action; stagnant agents do neither.
4. **Metabolic Intelligence:** The reframing of the Runtime from passive budget to active **Cognitive Control Policy**. Drawing on MIPS (Mitochondrial Information Processing System), we distinguish fast interventions (Apoptosis via mPTP-like triggers) from slow interventions (Retrograde Responses that reshape agent phenotype across sessions). The Runtime governs reasoning *quality* through the Metabolic-Epigenetic Coupling: low-budget states “methylate” expensive context, forcing efficient phenotypes.
5. **Multi-Cellular Organization:** The extension from single-agent to multi-agent systems using developmental biology. Agent phenotypes arise from differential context (epigenome) on shared weights (genome). Morphogen gradients (shared context variables) enable coordination without central control. Tissue boundaries enforce security isolation. This reframes “how many agents?” as “what is the developmental program?”
6. **Homeostasis:** Continuous self-repair through three modalities: Structural (Chaperone Loop with error-context feedback), Metabolic (Apoptosis + Regeneration with state summarization), and Cognitive (Autophagy via sleep/wake cycles). Most frameworks focus on Action; biology equally prioritizes Maintenance.
7. **Evolutionary Dynamics:** The Vermeij Trend predicts that agentic architectures face three selective pressures: adversarial robustness (Red Queen dynamics with attackers), task complexity (environmental pressure toward “aerobic” multi-step reasoning), and resource efficiency (metabolic selection toward the Pareto frontier). In our framing, these pressures motivate architectures that balance immune defense, task complexity, and metabolic regulation rather than treating those concerns as separable add-ons.
8. **Wire-Level Optics:** Prism optics enable conditional type-based routing (receptor specificity), Traversal optics enable batch processing (polymerase processivity), and DenatureFilters provide anti-injection sanitization—all composable on the same wire.
9. **Morphogen Diffusion:** A discrete-time dynamical system on the agent graph produces spatially varying concentration profiles, enabling position-dependent coordination without central control or global state sharing.
10. **Composable Coalgebras:** The coalgebraic formalism is made explicit with parallel and sequential composition, full transition traces, and observational equivalence criteria, enabling structured comparison of agent implementations.
11. **Epistemic Topology:** Kripke-style knowledge operators ( $K_i, E_G, C_G, D_G$ ) derived from wiring diagram structure yield four predictive theorems for multi-agent scaling—error amplification,

sequential penalty, parallel acceleration, tool density—and these theorems are qualitatively consistent with empirical results from large-scale architecture evaluations [22].

12. **Capability-Gated Tool Acquisition:** The Plasmid Registry implements Horizontal Gene Transfer with capability gating, preventing privilege escalation when agents dynamically acquire tools.
13. **Diagram Optimization via Categorical Rewriting:** Cost annotations enrich the wiring category over the resource monoid  $(\mathcal{R}, +, 0)$ , making resource consumption a first-class compositional concept. Rewriting passes—dead wire elimination, parallel grouping, cost-order scheduling—are endofunctors that preserve input-output observational equivalence while improving resource utilization. The ResourceAwareExecutor gates module execution on MetabolicState, connecting diagram optimization to the metabolic intelligence framework: under ATP depletion, non-essential pathway branches are downregulated, mirroring cellular starvation responses.

## The Mathematical Foundation

The correspondence rests on the category **Poly** of polynomial functors as a language for typed interfaces. Within that language, biological and software components at several scales—genes and agent capabilities, cells and agent runtimes, tissues and multi-agent subsystems—can be modeled as interfaces  $(O, I)$  consuming observations and producing actions. The Operad of Wiring Diagrams provides the grammar for composition, with type-checking at the topological level preventing classes of runtime errors. The Metabolic Coalgebra enriches this with resource constraints, providing a decidable termination criterion when costs are strictly decreasing and regeneration is excluded (or separately bounded). The Provenance Functor layers trust semantics onto message flow, providing a structural account of trust-gated resistance to content-level trust forgery.

## Implications

The Operon framework should be read as a structured design language for robust agent architectures. Its value is that some biological analogies can be made precise enough to guide implementation: topology constrains coordination, resource models constrain execution, and provenance labels constrain trust elevation.

The correspondence is therefore neither a loose metaphor nor a full biological equivalence. It is a disciplined abstraction that makes a subset of biologically inspired design patterns precise enough to analyze and implement, as demonstrated by the reference implementation.

The epistemic-topology formalization is qualitatively consistent with external scaling evidence such as Kim et al. [22], but it does not by itself constitute complete empirical validation of the framework. The connection between morphogen-driven topology switching and epistemic optimization instead suggests a path toward more formally analyzed adaptive multi-agent systems, where topological transitions can be justified under explicit epistemic and resource assumptions.

## The Six-Layer Arc

The six-layer progression demonstrates that the biological analogy extends beyond structural safety into temporal reasoning, adaptive behavior, and cognitive development:

1. **Structure:** Typed wiring diagrams, topology advice, pattern-first API.

2. **Memory:** Bi-temporal facts with dual time axes; auditable substrate integration with three-layer context model.
3. **Adaptation:** Pattern libraries for evolutionary template memory; watcher with three-category signal taxonomy (epistemic/somatic/species); intervention-count convergence signal grounded in Hao et al. [19]; experience-driven adaptive assembly.
4. **Cognition:** System A/B cognitive mode annotations per Dupoux et al. [13]; sleep consolidation with counterfactual replay; social learning with epistemic vigilance (trust-weighted template exchange); curiosity signals for intrinsic motivation.
5. **Development:** Critical periods that close as organisms mature; capability gating via developmental stage on tool acquisition; teacher-learner scaffolding.
6. **Integration:** Cross-subsystem integration tests; memory adapters bridging histone and episodic memory into the bi-temporal store; paper and documentation finalization.

Each layer assumes the previous one is stable. The progression from structural safety to temporal epistemics to adaptive cognition to developmental staging is not arbitrary—it mirrors the biological sequence from genome (fixed structure) through epigenetics (learned bias) to neural development (plastic then crystallizing).

## Future Work

Several directions remain open:

- **Convergence Investigation:** Combining Operon’s structural guarantees with operational runtimes (AnimaWorks [6], Swarms [31]) to create structurally guaranteed persistent organizations with cognitive capabilities.
- **Production Benchmarks:** Validation on real LLM outputs at scale via BFCL and AgentDojo evaluation harnesses, measuring both reliability and the impact of adaptive assembly on task performance.
- **Adversarial Robustness:** Red-team evaluation of immune evasion vectors and development of continuous adaptation mechanisms.
- **Distributed Diffusion:** Extending morphogen fields to distributed multi-process deployments with eventual consistency.
- **Learned Rewriting Rules:** Learning diagram optimization rules from execution traces, connecting to program synthesis and equality saturation.

## 12 Convergence: Integrating External Agent Frameworks

The preceding sections develop Operon’s structural analysis, epistemic topology, and formal verification foundations. This section summarizes how these tools extend to external agent orchestration systems through a typed adapter architecture. A standalone companion paper provides the full treatment, including implementation details, all 98 examples, and complete TLA+ specification listings; this section is self-contained but deliberately concise.

## 12.1 Five-Layer Architecture

The convergence architecture stacks five functional layers, each building on the one below. At the base, Operon provides the structural layer: typed wiring diagrams, epistemic theorems, and the resource-aware executor. Above it, three orchestration runtimes—Swarms [31] for enterprise-scale multi-agent workflows, DeerFlow [9] for session-based research pipelines, and Ralph [30] for hat-based event-driven agents—provide the operational execution surface. The thinking layer (AsyncThink) adds fork/join concurrency within individual stages, enabling speculative parallel reasoning without altering the outer topology. AnimaWorks [6] contributes the cognitive layer: developmental priming, heartbeat consolidation, and intrinsic motivation. At the top, A-Evolve [1] closes the loop with evolutionary skill refinement, where a Solve–Observe–Evolve–Gate–Reload cycle optimizes agent skills under monotonic score safety constraints.

A-Evolve — evolution layer
AnimaWorks — cognitive layer
AsyncThink — thinking layer
Ralph / DeerFlow / Swarms — orchestration layer
Operon — structural layer

## 12.2 ExternalTopology as Universal Intermediate Type

Every adapter produces an `ExternalTopology`—a frozen dataclass carrying the source framework tag, a pattern name, a tuple of agent specifications (plain dicts with at least `name` and `role` keys), directed communication edges, and arbitrary framework-specific metadata. The analysis pipeline is source-agnostic: `analyze_external_topology()` consumes any `ExternalTopology` and applies the four epistemic theorems (Theorems 4–7) to produce an `AdapterResult` containing topology advice, a suggested `PatternTemplate`, structural warnings, and a composite risk score. Adding a new orchestration target therefore requires writing a single parse function—no changes to the analysis code.

## 12.3 Epistemic Theorems as a Structural Linter

The epistemic theorems from §7 become a practical structural linter when applied through the adapter layer. Given an `ExternalTopology` with  $n$  agents, edge set  $E$ , and per-agent error probability  $p$ , the linter evaluates:

1. **Error Amplification** (Theorem 4): flags topologies where the aggregate failure probability exceeds a configurable tolerance.
2. **Sequential Penalty** (Theorem 5): warns when the critical-path length implies excessive latency or compounding error risk.
3. **Parallel Acceleration** (Theorem 6): identifies opportunities for parallelism that the current topology leaves unexploited.
4. **Tool Density** (Theorem 7): detects agents with tool sets exceeding the density threshold, recommending specialization.

For example, analyzing a Swarms `SequentialWorkflow` with five agents at  $p = 0.05$  per agent produces a sequential-penalty warning and suggests restructuring into a reviewer-gated parallel topology, reducing the expected error from  $1 - (1 - p)^n \approx 0.23$  to the parallel bound with reviewer verification.

## 12.4 TLA+ Verified Safety Invariants

Four TLA+ specifications verify safety invariants that are difficult to test exhaustively at the unit level:

1. **TemplateExchangeProtocol:** peer-to-peer template sharing preserves provenance, adopted templates carry correct source attribution, and trust scores update monotonically in the direction of observed outcomes.
2. **DevelopmentalGating:** stage transitions are irreversible (`EMBRYONIC`  $\rightarrow$  `JUVENILE`  $\rightarrow$  `ADOLESCENT`  $\rightarrow$  `MATURE`), critical-period windows close permanently, and tool acquisition respects the current stage’s minimum-stage constraint.
3. **ConvergenceDetection:** the watcher’s intervention-count ratio converges or triggers a `HALT` before exceeding the maximum intervention rate, preventing unbounded retry loops.
4. **EvolutionGating:** A-Evolve’s evolutionary loop maintains monotonic score safety—the deployed skill set’s fitness never regresses below the pre-evolution baseline.

Together, these specifications cover the critical safety boundaries of the convergence stack: trust integrity, developmental ordering, operational convergence, and evolutionary monotonicity.

## 12.5 Co-Design Convergence

The full adapter stack is formalized as a Zardini co-design problem [41]. Each adapter is a Design Problem (DP)—a monotone map from a resource poset (available agents, tools, context budget) to a functionality poset (task coverage, error bounds, latency). Series composition (`compose_series`) chains adapters where one DP’s functionality becomes the next’s resource; parallel composition (`compose_parallel`) runs independent DPs on disjoint resource slices. The adaptive assembly loop (`run`  $\rightarrow$  `record`  $\rightarrow$  `score`  $\rightarrow$  `select`) is feedback composition: a fixed-point iteration over the `PatternLibrary`’s scoring function. Convergence is guaranteed when the scoring function is monotone and the template space is finite—both hold by construction.

## 12.6 Adapter Summary

Table 5 summarizes the five adapters, their parse entry points, and the Operon concepts they map to.

## 12.7 Current Status

The convergence package comprises 14 Python modules (including the shared type definitions and the `__init__` re-exports) and 216 convergence-specific tests. The repository includes 98 examples (numbered 1–98), of which examples 86–98 demonstrate convergence features: adapter analysis (86–88), template exchange and hybrid assembly (89–91), memory bridging, priming views, heartbeat consolidation, and asynchronous thinking (92–95), co-design composition (96), and the Ralph and A-Evolve adapter workflows (97–98). All four TLA+ specifications pass model checking with TLC.

Table 5: Convergence adapters: framework, parse function, and Operon mapping.

Framework	Parse Function	Operon Mapping
Swarms [31]	<code>parse_swarm_topology()</code>	Workflow patterns → ExternalTopology, PatternTemplate seeding
DeerFlow [9]	<code>parse_deerflow_session()</code>	Session configs → ExternalTopology, bidirectional skill bridge
AnimaWorks [6]	<code>parse_animaworks_org()</code>	Org hierarchies → ExternalTopology, memory bridge to BiTemporalMemory
Ralph [30]	<code>parse_ralph_config()</code>	Hat definitions → ExternalTopology, cognitive-mode-annotated StageSpec
A-Evolve [1]	<code>parse_aevolve_workspace()</code>	Workspace manifests → ExternalTopology, evolved skill → StageSpec

## 12.8 Relationship to the Main Framework

The convergence layer depends on Operon’s internal packages (`core`, `patterns`, `memory`, `organelles`) but no core module depends on `operon_ai.convergence`. This one-way dependency ensures that the structural guarantees developed in the preceding sections remain valid independently of whether any external adapter is present. The adapters are consumers of the epistemic analysis API, not extensions of it.

For the complete treatment—including adapter implementation details, template exchange protocols, memory bridge semantics, TLA+ specification listings, and all 98 worked examples—see the standalone convergence companion paper.<sup>2</sup>

<sup>2</sup><https://coredipper.github.io/operon/convergence/>

## References

- [1] A-EVO-Lab. A-evolve: Autonomous llm agent evolution framework. <https://github.com/A-EVO-Lab/a-evolve>, 2025. Evolutionary framework for LLM agent skill optimization.
- [2] Vincent Abbott. Neural circuit diagrams: Robust diagrams for the communication, implementation, and analysis of deep learning architectures. *arXiv preprint arXiv:2402.05424*, 2024. Foundation for the diagrammatic scheme used in FlashAttention on a Napkin.
- [3] Vincent Abbott and Gioele Zardini. Flashattention on a napkin: A diagrammatic approach to deep learning io-awareness, 2025. Derives FlashAttention via Neural Circuit Diagrams with IO-aware performance model.
- [4] John F Allen. Energy transduction and the mind: mitochondria in brain function. *The Biochemist*, 44(4):8–13, 2022.
- [5] Uri Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450–461, 2007.
- [6] AnimaWorks Contributors. Animaworks: Autonomous agent runtime with priming and heartbeat consolidation. <https://github.com/AnimaWorks/AnimaWorks>, 2025. Operational runtime for autonomous agents with developmental priming.
- [7] B.J. Aubrey, A. Strasser, and G.L. Kelly. How does p53 induce apoptosis and how does this relate to p53-mediated tumour suppression? *Cell Death & Differentiation*, 25:104–113, 2018.
- [8] Michele Boreale. Coalgebras for bisimulation of weighted automata. *Logical Methods in Computer Science*, 19, 2023.
- [9] ByteDance Research. Deerflow: Deep exploration and efficient research flow. <https://github.com/bytedance/deer-flow>, 2025. Community-driven AI research workflow framework.
- [10] Navdeep S Chandel. Mitochondria as signaling organelles. *BMC Biology*, 12:34, 2014. Revisited in 2024 work on metabolic-epigenetic coupling.
- [11] Jesse Jiryu Davis. Reasoning about knowledge in TLA+. UCLA CS 201-A Seminar, 2026. MongoDB Distributed Systems Research Group.
- [12] Edoardo Debenedetti, Giorgio Severi, Nicholas Carlini, Scott Coull, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. In *NeurIPS*, 2024.
- [13] Emmanuel Dupoux, Yann LeCun, and Jitendra Malik. Why AI systems don’t learn and what to do about it. *arXiv preprint arXiv:2603.15381*, 2026.
- [14] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
- [15] Tongtong Feng, Xin Wang, and Wenwu Zhu. Self-evolving embodied AI. *National Science Review*, 2026.
- [16] Brendan Fong and David I Spivak. *Seven sketches in compositionality: An invitation to applied category theory*. Cambridge University Press, 2019.

- [17] Karl Friston. The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience*, 11(2):127–138, 2010.
- [18] Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.
- [19] Jie Hao, Xu Dai, Yuhang Qin, and Philip S. Yu. Brain-inspired graph multi-agent systems for LLM reasoning. *arXiv preprint arXiv:2603.15371*, 2026.
- [20] Dongming Jiang, Yi Li, Songtao Wei, Jinxin Yang, Dingyi Kang, Xu Hu, Ayushi Kishore, Qiannan Li, Alys Zhao, Feng Chen, and Bingzhe Li. Anatomy of agentic memory: Taxonomy and empirical analysis of evaluation and system limitations. *arXiv preprint arXiv:2602.19320*, 2026.
- [21] Christopher P Kempes, David H Wolpert, Zachary Cohen, and Juan Pérez-Mercader. The thermodynamic efficiency of computations made in cells across the range of life. *Philosophical Transactions of the Royal Society A*, 375(2109), 2017.
- [22] Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025. Google Research, Google DeepMind, and MIT.
- [23] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *ACM SIGMOD Record*, 41(3):34–43, 2012.
- [24] Minhua Lin, Zhiwei Zhang, Hanqing Lu, Hui Liu, Xianfeng Tang, Qi He, Xiang Zhang, and Suhang Wang. MemMA: Coordinating the memory cycle through multi-agent reasoning and in-situ self-evolution. *arXiv preprint arXiv:2603.18718*, 2026.
- [25] Michael Lynch and Georgi K Marinov. The bioenergetic costs of a gene. *Proceedings of the National Academy of Sciences*, 112(51):15690–15695, 2015.
- [26] Humberto R Maturana and Francisco J Varela. *Autopoiesis and cognition: The realization of the living*. Reidel, 1980.
- [27] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.
- [28] Georgi Nakov and Fredrik Nordvall Forsberg. Quantitative polynomial functors. In *Conference on Algebra and Coalgebra in Computer Science (CALCO)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- [29] Nelson Niu and David I Spivak. Polynomial functors: A mathematical theory of interaction. *arXiv preprint arXiv:2312.00990*, 2023.
- [30] Mikey O’Brien. Ralph: An event-driven agent orchestrator with hat-based role composition. <https://github.com/mikeyobrien/ralph-orchestrator>, 2025. Hat-based agent orchestration with backpressure and iteration limits.



- [31] Kye Gomez Osagie. Swarms: The enterprise-grade production-ready multi-agent orchestration framework. <https://github.com/kyegomez/swarms>, 2024. Multi-agent orchestration with sequential, concurrent, and graph workflows.
- [32] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.
- [33] Martin Picard and Orian S Shirihai. The mitochondrial information processing system: A new model of mitochondrial-nuclear communication. *Trends in Neurosciences*, 41(4):206–208, 2018. Revisited and expanded in 2022-2024 work.
- [34] Martin Picard and Orian S Shirihai. Mitochondria as multifaceted regulators of cell death. *Cell Metabolism*, 34(11):1607–1627, 2022.
- [35] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000. Foundational treatment of valid time, transaction time, and bi-temporal data models.
- [36] David I Spivak. Learners’ languages. *Compositionality*, 3(4), 2021.
- [37] Dmitry Vagner, David I Spivak, and Eugene Lerman. Algebras of open dynamical systems on the operad of wiring diagrams. *Theory and Applications of Categories*, 30(51):1793–1822, 2015.
- [38] Geerat J Vermeij. *The Evolution of Power: A New Understanding of the History of Life*. Princeton University Press, 2023.
- [39] Conrad H Waddington. *The Strategy of the Genes*. Allen & Unwin, 1957. Introduced the concept of the “epigenetic landscape”.
- [40] Jason Wei, Xuezhi Yi, Maarten Zhang, Yiming Liu, Xinyu Li, Xing Wang, Yujia Zhou, Yiming Li, Yuyang Wang, Zhi Li, et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022. Available at: <https://arxiv.org/abs/2201.11903>.
- [41] Gioele Zardini. *Co-Design of Complex Systems: From Compositionality to Monotone Theory*. PhD thesis, ETH Zurich, 2023. Monotone co-design theory for compositional system design.