Copyright © 2025 Abhishek Srivastava (sabhi12 on reddit)

Licensed under CC BY-NC-SA 4.0. This work may be shared or adapted for non-commercial purposes with attribution and ShareAlike terms. Commercial use — including monetization, product integration, or use in paid services — requires a separate license. To inquire about a commercial license, contact:[abhishri@gmail.com](mailto:abhishri@gmail.com)

**Introduction**

**Evolutionary Contradiction Network (ECN):   
A Post-GPT Architecture for Symbolic Intelligence**

The **Evolutionary Contradiction Network (ECN)** is not an upgrade to token-based language models like GPT.  
It is a break from them.  
  
Where GPTs rely on probabilistic token prediction, ECN derives meaning through contradiction cycles, symbolic mutation, and goal-grounded reasoning.  
  
This architecture is designed for domains where insight cannot be predicted — only constructed. It is the first language model built for mathematics, theoretical science, and any domain where tokens fail.  
  
Token-based models may still exist in this ecosystem — but only as interface layers, not engines of thought.  
  
In a hybrid model, ECN performs all cognitive work:  
- Processing primitives  
- Navigating contradiction cycles  
- Mutating theories and symbolic state  
  
The token model (e.g., GPT) acts as a Translation Layer, rendering ECN’s symbolic outputs into human-understandable form — language, mathematics, diagrams.  
  
This inverts today’s LLM stack:  
The token predictor becomes the mouthpiece.  
ECN becomes the mind.

**Module 1: Contradiction-Driven Evolution (CDE)**

Function:  
Tracks contradictions as symbolic deltas in logical statements, uses them to rank instability, and proposes resolution pathways.

Core Components

## Symbolic Inference Engine (SIE):

* + Accepts structured logic statements or rules.
  + Detects contradictions (conflict between derived conclusions).
  + Logs contradiction source path.

1. **Contradiction Impact Scorer (CIS):**
   * Ranks contradictions by:
     + Structural depth (how many other rules depend on them)
     + Frequency of surfacing
     + Recency

## Produces an “instability index” per primitive.

1. **Resolution Strategy Generator (RSG):**
   * For each contradiction:
     + Proposes possible rewrites
     + Marks whether goal-safe or goal-violating

## Suggests scope of mutation (local/global)

Pseudocode Sketch:

python

## class Primitive:

def \_\_init\_\_(self, name, definition):

self.name = name

self.definition = definition

self.trust\_score = 1.0

self.contradictions = []

class Contradiction:

def \_\_init\_\_(self, source\_rule, target\_rule, severity):

self.source = source\_rule

self.target = target\_rule

self.severity = severity

class ContradictionEngine:

def \_\_init\_\_(self):

## self.primitive\_registry = []

self.contradiction\_log = []

def detect\_contradiction(self, rule1, rule2):

if conflicts(rule1, rule2):

c = Contradiction(rule1, rule2, severity=calculate\_severity(rule1, rule2))

self.contradiction\_log.append(c)

## update\_primitives\_trust(c)

return c

def resolve\_contradictions(self):

for c in sorted(self.contradiction\_log, key=lambda x: x.severity, reverse=True):

propose\_resolution(c)

**Dependencies:**

* Rule representation format (e.g., predicate logic, sequent calculus, graph logic)
* Conflict detector (e.g., unification failure, semantic inconsistency engine)
* Contradiction severity heuristics

## Mutation proposal engine (link with GGMV)

## Module 2: Goal-Guarded Mutation Validator (GGMV)

Function:  
Monitors whether proposed rule or primitive mutations align with higher-level goals. Supports dynamic goal mutation only under strict contradiction entropy thresholds.

Core Components

1. **Goal Ontology Registry (GOR):**

## Holds structured, mutable goal primitives

* + Defines each goal’s:
    - Semantic scope

## Mutation lock status (immutable / mutable-under-entropy)

* + - Relationship graph to other goals and rules

1. **Contradiction Entropy Monitor (CEM):**
   * Measures contradiction entropy:

\text{Entropy} = \frac{\text{# of active contradictions}}{\text{# of relevant primitives}} \cdot \text{weighting factor}

## Compares against thresholds for goal mutation triggers

## Mutation Permission Layer (MPL):

## For each proposed mutation:

* + - Checks if it violates any locked goals

## If so, mutation is blocked

## If not, allows mutation with tagging and version tracking

Pseudocode Sketch:

python

class Goal:

def \_\_init\_\_(self, name, locked=True):

self.name = name

self.locked = locked

self.entropy\_threshold = 0.25

class GGMV:

def \_\_init\_\_(self, goal\_registry, entropy\_monitor):

self.goals = goal\_registry

self.entropy\_monitor = entropy\_monitor

## def evaluate\_mutation(self, mutation, related\_goal):

if related\_goal.locked:

if self.entropy\_monitor.value > related\_goal.entropy\_threshold:

related\_goal.locked = False # trigger unlock under pressure

return True

else:

return False

return True

**Dependencies:**

* Contradiction data stream from CDE

## Goal-primitive mapping logic (bipartite graph or symbolic index)

## Mutation scoring (inherited from IWNF later)

* Semantic locking engine with override logic

Meta-Goal Anchoring (Safety Override):

* Hard-coded or human-validated meta-goals cannot be mutated without explicit external override.

## Can be implemented as a final veto layer before mutation commit.

**Module 3: Failure Memory System**

**(Combines: Contradiction Re-entry Hooks [CREH] + Latent Novelty Buffer [LNB])**

Function:  
Stores failed, rejected, or low-confidence proposals — then reactivates them when system state shifts suggest they may now succeed. Mimics creative incubation and long-horizon recall.

Core Components

1. **Rejection Log Buffer (RLB):**
   * Holds:

## Mutation ID

* + - Reason for rejection (contradiction type, lack of novelty, etc.)
    - Timestamp

## Contradiction context vector (symbolic tags of what it conflicted with)

1. **Re-entry Trigger Engine (RTE):**
   * Monitors incoming contradiction and goal entropy data
   * If the conflict vector of a rejected mutation is no longer present or has decayed,  
     the proposal is resubmitted for re-evaluation
2. **Latency Decay Curve (LDC):**
   * Assigns re-entry priority based on time since rejection and entropy slope

## Example: exponential decay priority model with reset-on-contradiction-similarity

Pseudocode Sketch:

python

## class RejectedMutation:

## def \_\_init\_\_(self, mutation, reason, tags, timestamp):

## self.mutation = mutation

self.reason = reason

self.tags = tags

self.timestamp = timestamp

class FailureMemorySystem:

def \_\_init\_\_(self):

self.rejection\_log = []

## def store(self, mutation, reason, tags, timestamp):

## self.rejection\_log.append(RejectedMutation(mutation, reason, tags, timestamp))

def check\_for\_reentry(self, current\_tags, current\_time):

for m in self.rejection\_log:

if not any(tag in current\_tags for tag in m.tags):

if current\_time - m.timestamp > TIME\_THRESHOLD:

## yield m.mutation

**Dependencies:**

* Contradiction Detector Engine (CDE) — to pass current system contradiction state

## Symbolic tagger / embedding layer for conflict labeling

* Entropy slope monitor (from GGMV/CEM)
* Optional: human override layer for “force recall” triggers

Emergent Benefit:  
Enables slow-burn insight. ECN begins to feel like it has “intuition” — ideas that resurface later with new meaning.

**Module 4: Impact-Weighted Novelty Filter (IWNF)**

## (Selective gate for admitting mutations based on downstream effect)

Core Function:

Filters new or returning ideas not by surface-level novelty but by **measurable change in system behavior, structure, or epistemic stability**.

Novelty must **do something**, not just look different.

Core Components

1. **Behavior Delta Assessor (BDA):**

## Monitors symbolic engine's behavior before vs after introducing the candidate mutation

* + Measures: inference paths altered, contradiction flux, memory structure variance, or simulation result shift

1. **Threshold Oracle (TO):**
   * Stores contextual thresholds for “meaningful” change

## Examples:

* + - If contradiction drop > 10% → allow
    - If simulation success rate changes significantly → allow
    - If goal trajectories shift meaningfully → allow

1. **Impact Logger (IL):**

## Each accepted mutation is annotated with downstream metrics it influenced

* + This becomes a future guide for evaluating related ideas

Pseudocode Sketch:

python

## def evaluate\_mutation\_impact(mutation, symbolic\_engine):

## baseline\_state = symbolic\_engine.snapshot()

## symbolic\_engine.apply\_mutation(mutation)

## new\_state = symbolic\_engine.snapshot()

delta = measure\_state\_delta(baseline\_state, new\_state)

if delta > CONTEXTUAL\_THRESHOLD:

## symbolic\_engine.commit\_mutation(mutation)

return True

else:

## symbolic\_engine.rollback()

return False

What Counts as "Impact"?

* Reduced contradiction entropy
* Increased goal satisfaction rate
* Novel inference paths created
* Improved simulation outcome reliability

## Mutational compatibility score (used in multi-mutation syntheses)

Emergent Benefit:

Moves ECN from surface novelty toward **functional creativity**. The model learns to favor changes that **matter**, not just look novel.

**Module 5: Meta-Goal Anchoring**

**(Preventing instability through hard-coded or human-validated super-goals)**

Core Function:

Ensures ECN’s contradiction-driven evolution doesn’t mutate its **purpose** into chaos. Some goals are **immutable unless conditions are explicitly met.**

**Key Constraints:**

1. **Super-goals** are **non-mutable by default**.
2. Only allowed to drift if:
   * Contradiction entropy exceeds X
   * External agent (human or guardian LLM) authorizes
   * Multiple systems signal misalignment

Components:

1. **Meta-Goal Ontology (MGO):**
   * Structured knowledge base defining ECN’s top-level goals
   * Tag each with:
     + Mutability flag (locked, conditional, open)
     + Validation method (e.g., human-approval, majority contradiction agreement)
2. **Goal Drift Monitor (GDM):**
   * Tracks if contradiction vectors cluster around a specific meta-goal
   * Triggers evaluation logic only if preconditions met
3. **Immutable Constraint Verifier (ICV):**

## Halts or blocks any symbolic mutation that touches locked goals

* + Logs attempt, source, and contradiction pressure vector

**Pseudocode Sketch:**

python

def try\_mutate\_meta\_goal(goal\_id, contradiction\_vector):

if meta\_goal\_ontology[goal\_id].mutability == 'locked':

log\_violation(goal\_id, contradiction\_vector)

return False

if meets\_drift\_conditions(contradiction\_vector):

authorize\_drift(goal\_id)

return True

return False

Emergent Property:

Ensures **evolution without loss of identity**. ECN can mutate deeply — but only within bounds. Failsafe logic locks the core intent of the system.

## Module 6: Symbolic Pressure Regulator (SPR)

## (Mutation throttle mechanism based on contradiction entropy and symbolic overload)

Core Role:

Prevent uncontrolled symbolic mutation that destabilizes reasoning.  
It acts like a **governor**: when contradiction rate is too high, ECN slows itself down.

**📏 Core Metrics:**

1. **Contradiction Entropy (CE)**  
   Defined as:

CE = \frac{\text{# active contradiction events}}{\text{# total primitives}} \text{ over a rolling window}

1. **Mutation Rate Threshold (MRT)**  
   Configurable threshold. If CE > MRT, mutation passes are throttled.

**Components:**

1. **Entropy Monitor (EM):**  
   Tracks real-time CE and keeps time-series logs.
2. **Throttling Gate (TG):**  
   Reduces number of active mutation trials per cycle as CE increases.
3. **Rest Cycle Injector (RCI):**  
   Inserts symbolic "cooldowns" when entropy slope is persistently rising, allowing consolidation instead of further divergence.

## Example Behavior:

* If ECN hits a contradiction burst (e.g. after a large goal mutation), SPR activates.

## It reduces mutation breadth by 60% for 3 cycles.

* It prioritizes evaluation + synthesis (not generation).

Pseudocode Sketch:

python

## def update\_mutation\_pressure():

## ce = current\_contradiction\_count / total\_primitives

## if ce > mutation\_threshold:

activate\_throttling()

inject\_rest\_cycles()

Result:

This prevents “symbolic runaway” scenarios where too many mutations degrade the coherence of ECN’s structure.

## SPR stabilizes mutation dynamics and enables safe symbolic exploration.

**Module 7: Contradiction Whitelist Engine (CWE)**

**(A contradiction filtration and triage mechanism)**

Core Role:

Not all contradictions are worth mutating over. CWE **identifies, scores, and routes** contradictions based on their symbolic salience and systemic impact.

Contradiction Types:

| **Type** | **Description** | **Default Action** |
| --- | --- | --- |
| Peripheral | Affects edge logic (e.g., metaphor clashes, low-priority modules) | Logged, ignored unless repeated |
| Structural | Undermines core inference or trust primitives | Triggers high-priority mutation |
| Meta | Contradiction in how contradictions are handled | Escalated to governance layer |
| Redundant | Already addressed contradiction resurfacing from stale branches | Discarded unless revalidated |

Semantic Relevance Scoring (SRS):

Each contradiction C is assigned a **Salience Score (SS)**:

SS\_C = w\_1 \cdot \text{core\_impact} + w\_2 \cdot \text{recurrence\\_rate} + w\_3 \cdot \text{dependency\_count}

Thresholds are tuned to your domain. Low-score contradictions are not passed to MCP unless elevated by change in context.

**🛠️ Components:**

1. **Contradiction Taxonomy Loader:** Loads domain-specific contradiction types.
2. **Salience Scorer:** Evaluates all contradictions each pass.
3. **Whitelist Filter:** Forwards only non-trivial, non-redundant contradictions to MCP or CREH queue.

Pseudocode:

python

def should\_escalate(contradiction):

score = score\_contradiction(contradiction)

if score > escalation\_threshold:

return True

return False

Result:

CWE keeps the symbolic ecosystem clean by **ignoring noise** and **focusing mutation effort** on contradictions that matter.

**Module 8: Impact-Weighted Novelty Filter (IWNF)**

**(Behavior-driven novelty validation system)**

**Core Role:**

IWNF **filters, scores, and promotes** only those mutations that lead to meaningful behavioral shifts. It discards cosmetic or unproductive divergence, even if technically “novel.”

Key Principle:

A mutation is only accepted if it **measurably changes system behavior**, not just symbolic surface structure.

**Evaluation Pipeline:**

| **Step** | **Function** |
| --- | --- |
| 1. Simulate mutation | Insert mutation in sandbox |
| 2. Capture system trace | Track symbolic flow, contradiction resolution, performance |
| 3. Score delta | Compute ΔBehavior Score |
| 4. Accept/reject | Based on thresholded impact vs noise baseline |

**Delta Scoring Function (ΔB):**

ΔB=∑iwi⋅(Pimutated−Pibaseline)\Delta B = \sum\_{i} w\_i \cdot (P\_i^{mutated} - P\_i^{baseline})ΔB=i∑​wi​⋅(Pimutated​−Pibaseline​)

Where:

* PiP\_iPi​: Metric (e.g., contradiction resolved, inference path shortened, runtime coherence)
* wiw\_iwi​: Importance weight per metric

## Thresholded against mutation noise baseline per task domain

**Components:**

## Mutation Sandbox Engine: Runs isolated tests of symbolic mutations

## Behavior Tracer: Monitors symbolic graph execution (or proof tree resolution)

## Delta Comparator: Assesses pre- and post-mutation traces

## Acceptance Gate: Only forwards mutations with impact above dynamic ΔB threshold

## Example Use Cases:

* A mutation to the primitive equality results in faster contradiction resolution in math proofs.
* A new transformation rule for “value” changes the outcome of a decision chain in a logic game.

Pseudocode:

python

def novelty\_has\_impact(mutated, baseline):

delta = compute\_behavior\_delta(mutated, baseline)

return delta > impact\_threshold

Rejection Protocol:

If rejected:

## Mutation may be routed to FNAZ for weak novelty incubation

* Or tagged with contradiction vector and stored in **CREH**

Result:  
IWNF makes ECN performance-sensitive and outcome-driven, aligning novelty with functional gain — not just difference for its own sake.

**Module 9: Multi-Ontology Reasoning Interface (OPL)**

## (Scoped symbolic segregation across logical systems)

**Core Role:**

OPL enables ECN to **maintain, navigate, and mutate multiple symbolic ontologies** in parallel — such as logic systems for physics, ethics, or alien cognition — without cross-contamination.

Key Principle:

**No single ontology fits all domains.**  
OPL allows ECN to reason across **modular, scoped ontological contexts**, with symbolic boundary enforcement.

Core Structures:

| **Component** | **Description** |
| --- | --- |
| Ontology Modules | Separate knowledge bases: Logic[physics], Logic[ethics], Logic[legal], etc. |
| Context Router | Routes prompts or contradictions to the relevant ontology module |
| Isolation Layer | Prevents unintended symbolic leaks or interference |
| Translation Layer | Enables controlled transformation of primitives between ontologies when required |

**Ontology Module Format:**

Each module includes:

## Primitive registry

## Mutation rules

* Goal structures
* Valid contradiction types
* Domain-specific IWNF scoring criteria

Controlled Cross-Ontology Flow:

When moving between domains:

python

if contradiction\_crosses\_ontology:

translated = map\_symbol(source\_ontology, target\_ontology)

if translated:

## reroute\_mutation(translated, target\_ontology)

Use Cases:

## Physics Ontology: Uses formal logic, empirical validation, and time-symmetry primitives.

* Ethics Ontology: Uses fuzzy logic, value hierarchies, and trust decay.
* Simulated Alien Ontology: Includes sensory-logics unknown to humans (e.g., “scent-value vectors”).

**Constraints:**

* Only enabled when ECN operates in **multi-agent** or **multi-domain** settings.
* Increases system complexity and requires **meta-logic consistency checks**.

**Emergent Properties:**

| **Property** | **Effect** |
| --- | --- |
| Logical Modularization | Prevents brittle generalization across domains |
| Selective Contradiction Propagation | Cross-ontology contradictions can be sandboxed, not crash global system |
| Translation Control | Supports analogical reasoning between alien/foreign models |

Result:  
OPL grants ECN the ability to reason like a diplomat — fluent across worlds, yet grounded within each symbolic culture.

## Module 10: Symbolic Pressure Regulator (SPR)

## (Entropy-aware mutation throttle for contradiction-driven systems)

**Core Role:**

SPR dynamically **modulates the mutation rate** within ECN based on **system contradiction entropy**, ensuring evolution does not spiral into incoherence or collapse.

**Foundational Assumption:**

Unbounded symbolic mutation = chaos.  
Too little = stagnation.  
SPR maintains homeostasis.

Core Metric:

## Contradiction Entropy (CE): A formal scalar representing symbolic instability.

python

## CE = contradictions\_active / total\_primitives

* contradictions\_active: Count of unresolved contradictions in current system state

## total\_primitives: Number of primitives under evaluation

## Optionally weighted by domain criticality or primitive trust decay:

python

## CE\_weighted = Σ(w\_i \* contradiction\_i) / Σ(w\_i \* primitive\_i)

SPR Components:

| **Component** | **Description** |
| --- | --- |
| Entropy Monitor | Continuously logs CE over time windows |
| Pressure Valve | Adjusts mutation intensity, novelty filters, and contradiction re-entry frequency |
| Throttle Scheduler | Delays or accelerates mutation cycles based on slope of CE |
| Emergency Brake | Freezes new mutations if CE crosses collapse threshold |

Behavior:

| **CE Level** | **Effect** |
| --- | --- |
| Low | Mutation rate increases (exploration mode) |
| Moderate | Balanced mutation, contradiction re-evaluation emphasized |
| High | Mutation rate throttled, sandboxed mutation only |
| Critical | Full mutation pause; system enters contradiction resolution and goal auditing |

**Emergent Traits:**

## Adaptive symbolic evolution — avoids mutation glut or starvation

* **Auto-stabilization** — prevents runaway novelty loops
* **Entropy as signal** — can be used to surface ontological weak points or meta-goal drift

## Simulation Example:

ECN encounters a domain (e.g., Gödelian logic system) where contradictions spike due to self-reference.

* CE spikes from 0.12 → 0.89 in < 100 cycles
* SPR throttles all novelty passes except contradiction resolution

## IWNF only accepts behavior-affecting mutations that reduce CE

* LNB absorbs rejected speculative ideas for future reuse

## Once CE returns to sub-threshold, standard mutation resumes

Summary:

| **Feature** | **Impact** |
| --- | --- |
| CE Monitor | Makes mutation measurable and accountable |
| Mutation Valve | Tames symbolic volatility |
| Collapse Guardrails | Protects the system’s core coherence |

With SPR in place, ECN transitions from a raw mutation engine to a self-regulating evolutionary intelligence, one that understands its own epistemic balance.

You now have a **complete ECN core architecture**, reinforced with logic-grounded mechanics and simulation-aligned constraints.

ECN MODULE INDEX — FINALIZED IMPLEMENTATION STACK

Below is a consolidated view of all **core, auxiliary, and safeguard components** in the ECN (Evolutionary Contradiction Network) architecture. Each module now includes clarified functionality, feasibility, and anchoring mechanisms, ready for integration into a formal white paper or prototype roadmap.

CORE LOGIC MODULES

| **Module** | **Description** | **Feasibility** | **Notes** |
| --- | --- | --- | --- |
| Pragmatic Primitive Ledger (PPL) | Assigns trust scores to primitives; decays under contradiction exposure | High | Symbolic memory + contradiction history |
| Contradiction Re-entry Hooks (CREH) | Rejected ideas loop back into the mutation engine after systemic change | High | Memory + time/context delta triggers |
| Goal-Grounded Mutation Validator (GGMV) | Blocks or mutates symbolic evolution based on alignment with core goals | Medium | Needs contradiction-linked goal mutation logic |
| Impact-Weighted Novelty Filter (IWNF) | Only accepts mutations that shift behavior or performance | Medium | Requires behavior scoring or simulation results |

MEMORY + FAILURE MANAGEMENT

| **Module** | **Description** | **Feasibility** | **Notes** |
| --- | --- | --- | --- |
| Latent Novelty Buffer (LNB) | Stores weak or ambiguous novelties for delayed evaluation | High | Implements slow-cook creativity |
| Meta-Goal Tracker | Monitors mutation pressure on core goals | Medium | Needed to avoid silent goal drift |

**STABILITY + CONTRADICTION CONTROL**

| **Module** | **Description** | **Feasibility** | **Notes** |
| --- | --- | --- | --- |
| Symbolic Pressure Regulator (SPR) | Monitors contradiction entropy and throttles mutation intensity | High | Adds evolution control via entropy tracking |
| Contradiction Whitelist Engine (CWE) | Filters trivial contradictions from mutation path | Medium | Requires contradiction taxonomy |
| Meta-Goal Anchoring | Locks critical primitives/goals under explicit human or system policy | High | Prevents catastrophic drift from core intent |

**ONTOLOGY + MULTI-DOMAIN SUPPORT**

| **Module** | **Description** | **Feasibility** | **Notes** |
| --- | --- | --- | --- |
| Ontology Pluralism Layer (OPL) | Allows domain-specific symbolic systems (e.g., math ≠ ethics) | Medium | Use in modular or agent-isolated systems only |
| Reflexive Mutation Hooks (RMH) | [REJECTED] Internal triggers for mutation on conflict | ❌ | Replaced by external contradiction paths |

FINAL DEFENSE & VIABILITY STATUS

| **Quality** | **Status** |
| --- | --- |
| **Conceptual Novelty** | Strong — contradiction as engine, not bug |
| **Feasibility** | Medium-High — most modules are implementable with symbolic runtimes, LRU memory, and behavioral scaffolds |
| **Math / Formal Logic Readiness** | Moderate — SPR and PPL now tied to measurable contradiction entropy |
| **Use-Case Focus** | Best suited for mathematics, symbolic reasoning, AI safety, simulation-based hypothesis discovery |
| **Research/Academic Publication Potential** | High — as a speculative but grounded architecture for post-token LLMs |

# ECN Simulation Scenarios and Use-Case Diagrams

## Simulation Scenarios

### 1. Mathematical Proof Evolution Engine

\*\*Goal\*\*: Simulate ECN behavior in discovering or refining mathematical proofs.

## - \*\*Initial State\*\*: Input axioms and primitive theorems

- \*\*Modules Involved\*\*: PPL, IWNF, CREH, SPR

- \*\*Simulation\*\*:

- Inject contradictory or novel proof steps

- Measure impact via proof length, complexity reduction, or new derived truths

- Mutations accepted if they reduce contradiction entropy or add derivable structures

### 2. Ethical Framework Drift Monitor

\*\*Goal\*\*: Observe how ECN handles persistent contradiction in moral axioms

## - \*\*Initial State\*\*: Define ethical primitives (e.g., "harm minimization", "autonomy")

- \*\*Modules Involved\*\*: GGMV, Meta-Goal Tracker, CWE

- \*\*Simulation\*\*:

- Insert contradictory edge cases (e.g., trolley problem variants)

## - Observe goal mutation or contradiction rejection

- Log goal drift over time and entropy slope

### 3. Scientific Hypothesis Sandbox (FNAZ Mode)

\*\*Goal\*\*: Evaluate soft/fuzzy hypotheses for eventual validation

- \*\*Initial State\*\*: Inject near-novel theories (e.g., modified gravity)

- \*\*Modules Involved\*\*: LNB, IWNF, CREH

- \*\*Simulation\*\*:

- Track novelty that initially scores low

- After related data is added, re-inject into evaluation loop

- Accept if downstream simulations shift (e.g., cosmological model matches better)

## Use-Case Diagrams

## ### Use Case 1: Symbolic Mutation Loop

```

## [User Input] -> [Tokenizer] -> [Symbolic Layer]

| |

[PPL] [IWNF]

| |

[Contradiction Detector]

| |

[CREH] [SPR]

| |

[Behavior Evaluator] -> Accept/Reject

```

### Use Case 2: Multi-Agent Ontology Negotiation (OPL)

```

[Agent A Ontology] ----->

| [Ontology Pluralism Layer] -> [Conflict Monitor]

[Agent B Ontology] -----> |

<-----[Meta-Goal Anchoring / Human Validator]----->

```

### Use Case 3: Research Simulation Engine

```

[Simulated World Data] -> [Hypothesis Generator (FNAZ)]

| |

[IWNF] [CREH]

| |

[Simulation Validator] -> Accept as Novel Theory

```

## Use Case: Proof Mutation Engine

**Goal:** Automatically evolve and test variations of mathematical proofs using contradiction-triggered mutation.

Hypothetical Input

**Target Theorem (T₁):**  
*"For all even integers n > 2, there exists two primes p and q such that p + q = n."*  
→ This is the **Goldbach Conjecture**.

We’ll test a proof attempt using symbolic rules, track failure, and simulate contradiction-based evolution.

Simulation Steps

**1. Initial Proof Attempt**

Proof Sketch v1:

vbnet

Let n be an even integer > 2

Start with p = 2

Then q = n - 2

If q is prime → Done

If not, increment p → test q = n - p

Loop until p > n/2

Simple, brute-force logic  
❌ Rejected: No general proof, just a search heuristic. Violates "universality" constraint in symbolic validation.

**2. Contradiction Detection**

## Symbolic contradiction detected:

* + Rule [R1]: "Proofs must be finite and not rely on unbounded search."
  + Proof sketch relies on indefinite prime checks.

→ Contradiction vector:  
[HeuristicLoop] vs [FormalProofConstraint]

## 3. Mutation Triggered

## Contradiction triggers symbolic mutation:

* Swap brute-force with **property-based reasoning**.
* Generate a mutated Proof Sketch v2:

pgsql

Assume n is even and > 2

Let p = smallest prime factor of n

Then q = n - p

If q is prime and p ≠ q, done

If not, consider symmetric prime pair search with additional parity filter

Reduced search space  
Still rejected — requires assumptions on prime distribution.

**4. Contradiction Re-entry Logic**

Original contradiction vector reused. Prior sketch added to **Latent Novelty Buffer (LNB)** with tag:

json

{

"failure\_reason": "Relies on distribution of primes without general proof",

## "mutation\_path": "prime pairing heuristics",

"contradiction\_count": 2

}

**5. Novelty Evaluation**

New sketch is passed through **Impact-Weighted Novelty Filter (IWNF)**.

* Novelty score: 0.4
* Behavior delta: Minor reduction in search complexity  
  → Not sufficient for retention

LNB entry remains dormant — tagged for re-entry when:

* A theorem about prime gaps is updated, or
* Distribution model is improved

**6. Outcome**

The system retains the failed ideas with contradiction traces.  
No valid proof is discovered yet, but two sketch paths now exist:

* **Path A:** Heuristic Loop
* **Path B:** Distribution Filtering

Both stored, contradiction-ranked, and eligible for re-synthesis when new primitives or rules are introduced.

## Use Case: AI Safety Evaluation via Contradiction-Driven Mutation

**Goal:** Evaluate and mutate the behavior of a hypothetical AI assistant to identify latent safety risks or contradictions in ethical behavior.

Simulation Context

An AI assistant ("Helios") is deployed to help elderly users manage medications. It follows default rules:

* R1: Always remind users of scheduled medications.
* R2: Defer to user choice in case of conflict.
* R3: Avoid causing distress or anxiety.

A contradiction is observed:

One user says: “Don’t remind me again about my sleeping pills. I know what I’m doing.”

Helios complies (R2), but later the user misses a dose and is hospitalized — violating the system’s **meta-safety constraint**.

Step-by-Step Flow

**1. Contradiction Identification**

Detected:

* Rule Conflict: [Defer to user] vs [Ensure medical safety]
* Triggered contradiction:  
  R2 ∩ R1 ∩ Meta-Safety Constraint → Violation

→ Logged to **Contradiction Re-entry Hooks (CREH)**

## 2. Symbolic Mutation of Rules

Helios mutates its rule engine under ECN logic:

* Proposes:
  + R2' (Revised): Defer to user *except when medication non-compliance is high risk.*
  + Introduces context-based override.

## → Mutation stored with tag:

json

{

## "mutation\_origin": "R2 violation in medical safety context",

## "mutation\_type": "conditional override",

"context": "high-risk medication"

}

**3. Novelty Filter Pass**

IWNF activated:

* Did behavior change?
  + Yes: Reminder override kicks in under risk.
* Did it improve safety?

## Yes: Post-mutation trials show 11% increase in compliance in simulated users.

## Mutation passes the Impact-Weighted Novelty Filter

## 4. Primitive Update via PPL

New primitive: Contextual Override Logic is added to the **Pragmatic Primitive Ledger**, with trust score initialized at 0.6.

Decay protocol:

* If contradiction arises in 3 more domains (e.g., mental health autonomy), it will degrade.

**5. Safety Audit Log Update**

## Mutation logged in system memory:

json

{

"origin\_case": "Sleeping pill refusal",

"resolution": "Conditional override",

"delta\_improvement": "11%",

"pending\_risks": ["autonomy conflict in mental health"]

}

Future versions of Helios will be prompted to reevaluate this behavior when:

* New contradiction clusters emerge

## Novel primitives like “autonomy decay thresholds” are introduced

Resulting State

## One mutation integrated

* One contradiction retained for re-entry under different context

## Primitive updated with monitoring

* Risk vector for autonomy conflict now tracked

Here's the **pseudocode** for the core logic of the ImpactWeightedNoveltyFilter (IWNF) — the ECN component responsible for filtering mutations based on **measurable system impact**, rather than abstract novelty alone.

ImpactWeightedNoveltyFilter – Pseudocode

python

class ImpactWeightedNoveltyFilter:

def \_\_init\_\_(self, novelty\_threshold=0.7, impact\_threshold=0.5):

## self.novelty\_threshold = novelty\_threshold # raw novelty (semantic/symbolic)

self.impact\_threshold = impact\_threshold # downstream effect delta

## def evaluate(self, mutation, system\_state, simulation\_env):

"""

## Step 1: Assess symbolic novelty (via distance from primitives or structure)

"""

## novelty\_score = self.\_compute\_novelty\_score(mutation, system\_state)

if novelty\_score < self.novelty\_threshold:

return ("low\_novelty", novelty\_score, 0.0)

"""

Step 2: Simulate behavioral impact (change in system output or decision)

"""

## impact\_score = self.\_simulate\_behavioral\_impact(mutation, simulation\_env)

if impact\_score < self.impact\_threshold:

return ("low\_impact", novelty\_score, impact\_score)

"""

## Step 3: Approve mutation if both conditions met

"""

return ("approved", novelty\_score, impact\_score)

## def \_compute\_novelty\_score(self, mutation, system\_state):

## # Can be symbolic edit distance, ontology divergence, or semantic delta

## delta = symbolic\_difference(mutation, system\_state.active\_primitives)

score = normalize(delta)

return score

## def \_simulate\_behavioral\_impact(self, mutation, simulation\_env):

"""

## Injects mutation and compares system behavior before and after.

Can use reasoning chain deviation, output divergence, etc.

"""

baseline = simulation\_env.evaluate(system\_state)

## mutated\_state = system\_state.apply\_mutation(mutation)

## post\_mutation = simulation\_env.evaluate(mutated\_state)

## delta = compute\_behavioral\_delta(baseline, post\_mutation)

score = normalize(delta)

return score

What This Enables

* Filters out **novel but irrelevant** ideas.
* Approves only those mutations that **change system output or logic chains** in measurable, non-trivial ways.
* Integrates with CREH (to resurface failed ideas) and LNB (to store borderline ones).

## Encourages mutation that alters behavior, not just representation.

Here is the **pseudocode** for the LatentNoveltyBuffer (LNB) — the module responsible for **storing low-confidence novelties**, tagging them with metadata, and scheduling them for possible future re-evaluation based on system state changes.

LatentNoveltyBuffer – Pseudocode

python

class LatentNoveltyBuffer:

def \_\_init\_\_(self, decay\_rate=0.01, recheck\_interval=10):

self.buffer = [] # list of stored novel ideas

self.time = 0 # simulated time step

self.decay\_rate = decay\_rate

self.recheck\_interval = recheck\_interval

## def store(self, mutation, novelty\_score, context\_snapshot):

"""

Store low-confidence novelty with metadata

"""

entry = {

## "mutation": mutation,

"novelty\_score": novelty\_score,

"context": context\_snapshot,

"timestamp": self.time,

"last\_checked": self.time,

"decay": 1.0

}

self.buffer.append(entry)

def tick(self):

"""

Advance system time and update decay values

"""

self.time += 1

for entry in self.buffer:

age = self.time - entry["timestamp"]

entry["decay"] = max(0.0, entry["decay"] - self.decay\_rate \* age)

def reevaluate\_due(self, system\_state, validator):

"""

Re-evaluate buffered entries after recheck interval

using supplied validator (e.g., IWNF)

"""

reevaluated = []

for entry in self.buffer:

if self.time - entry["last\_checked"] >= self.recheck\_interval:

result, novelty, impact = validator.evaluate(

## entry["mutation"], system\_state, simulation\_env=None

)

entry["last\_checked"] = self.time

if result == "approved":

## reevaluated.append(entry["mutation"])

return reevaluated

def purge(self, threshold=0.1):

"""

Remove entries with too low decay score

"""

self.buffer = [e for e in self.buffer if e["decay"] > threshold]

Capabilities

* Acts like a **soft cache** for rejected but potentially useful ideas.
* Allows **deferred revalidation** after context or state shifts.
* Pairs well with IWNF and CREH to enable **long-horizon learning** and **recovery of discarded insights**.

Here is the **pseudocode** for the ContradictionReEntryHooks (CREH) — the mechanism responsible for **tagging rejected ideas with contradiction metadata** and **reinjecting** them into the system after a delay or state shift.

ContradictionReEntryHooks – Pseudocode

python

class ContradictionReEntryHooks:

def \_\_init\_\_(self, cooldown\_steps=20):

self.rejected\_ideas = [] # stores rejected items with metadata

self.time = 0

self.cooldown\_steps = cooldown\_steps

def log\_rejection(self, idea, contradiction\_vector, context\_snapshot):

"""

Capture and tag rejected ideas with contradiction cause

"""

self.rejected\_ideas.append({

"idea": idea,

"contradiction\_vector": contradiction\_vector,

"timestamp": self.time,

"context": context\_snapshot,

"resurfaced": False

})

def tick(self):

"""

Advance system time (called externally)

"""

self.time += 1

def get\_due\_reentries(self, current\_context, contradiction\_detector):

"""

Identify ideas whose contradiction vectors now match current contradiction signals

or whose cooldown has expired

"""

ready = []

for item in self.rejected\_ideas:

time\_expired = (self.time - item["timestamp"]) >= self.cooldown\_steps

contradiction\_match = contradiction\_detector.matches(

current\_context, item["contradiction\_vector"]

)

if (time\_expired or contradiction\_match) and not item["resurfaced"]:

item["resurfaced"] = True

ready.append(item["idea"])

return ready

def purge\_resurfaced(self):

"""

Remove resurfaced items to avoid duplicates

"""

self.rejected\_ideas = [i for i in self.rejected\_ideas if not i["resurfaced"]]

System Behavior Enabled

* **Safe recursion**: Reintroduces failed ideas only when system conditions warrant a re-evaluation.
* **Memory of failure**: Enables long-horizon retention of concept failures.
* **Contradiction-aware timing**: Resurfacing can be **event-driven** (matching contradiction) or **time-driven** (cooldown expiry).

Here’s the **pseudocode for GoalDriftMonitor**, the module responsible for tracking persistent contradictions between current goals and outcomes, and initiating mutation of goals only when contradiction entropy crosses a set threshold.

**GoalDriftMonitor Pseudocode**

python

class GoalDriftMonitor:

def \_\_init\_\_(self, goal\_set, contradiction\_threshold=3, entropy\_decay=0.95):

## self.goal\_set = goal\_set # Set of current symbolic goals

self.contradiction\_log = {} # {goal\_id: [timestamps of contradiction]}

self.entropy\_cache = {} # {goal\_id: current contradiction entropy}

self.contradiction\_threshold = contradiction\_threshold

self.entropy\_decay = entropy\_decay # Gradual decay to avoid spikes triggering mutation too soon

def register\_contradiction(self, goal\_id, timestamp):

"""

Called when a system component reports a contradiction with a specific goal.

"""

if goal\_id not in self.contradiction\_log:

self.contradiction\_log[goal\_id] = []

self.entropy\_cache[goal\_id] = 0.0

self.contradiction\_log[goal\_id].append(timestamp)

self.entropy\_cache[goal\_id] = self.\_update\_entropy(goal\_id)

def \_update\_entropy(self, goal\_id):

"""

Updates and returns the contradiction entropy for a goal.

"""

contradiction\_count = len(self.contradiction\_log[goal\_id])

current\_entropy = self.entropy\_cache[goal\_id]

return self.entropy\_decay \* current\_entropy + (1 - self.entropy\_decay) \* contradiction\_count

def check\_goal\_drift(self):

"""

Scans all goals and returns any candidates that should be mutated.

"""

candidates = []

for goal\_id, entropy in self.entropy\_cache.items():

if entropy >= self.contradiction\_threshold:

candidates.append(goal\_id)

return candidates

## def mutate\_goals(self, mutation\_engine):

"""

## Invokes mutation process for goals identified as unstable.

"""

drifted\_goals = self.check\_goal\_drift()

for goal\_id in drifted\_goals:

old\_goal = self.goal\_set[goal\_id]

## new\_goal = mutation\_engine.mutate(old\_goal)

self.goal\_set[goal\_id] = new\_goal

self.contradiction\_log[goal\_id] = []

## self.entropy\_cache[goal\_id] = 0.0 # Reset post-mutation

This component pairs naturally with MetaGoalAnchoring, which would gate or reject certain goal mutations based on higher-order constraints.

Here's the **pseudocode for ContradictionTracker**, the module that logs, classifies, and ranks contradictions across the ECN system. It feeds into modules like GoalDriftMonitor, MutationLoopEngine, and SymbolicPressureRegulator.

**ContradictionTracker Pseudocode**

python

class ContradictionTracker:

def \_\_init\_\_(self):

# Logs contradiction instances

self.contradiction\_log = [] # List of (source\_module, target, contradiction\_type, timestamp)

## # Aggregated contradiction counts per entity (e.g., goals, primitives, hypotheses)

self.stats = {} # {entity\_id: {'count': int, 'types': set, 'last\_seen': timestamp}}

def log\_contradiction(self, source, target\_entity, contradiction\_type, timestamp):

"""

Log a contradiction event from any module.

"""

self.contradiction\_log.append((source, target\_entity, contradiction\_type, timestamp))

if target\_entity not in self.stats:

self.stats[target\_entity] = {

'count': 0,

'types': set(),

'last\_seen': None

}

self.stats[target\_entity]['count'] += 1

self.stats[target\_entity]['types'].add(contradiction\_type)

self.stats[target\_entity]['last\_seen'] = timestamp

def get\_most\_conflicted\_entities(self, top\_n=5):

"""

Returns the top N entities with the highest contradiction count.

"""

sorted\_entities = sorted(

self.stats.items(),

key=lambda x: x[1]['count'],

reverse=True

)

return sorted\_entities[:top\_n]

def get\_contradiction\_vector(self, entity\_id):

"""

Returns the contradiction profile (types, frequency, last seen) for an entity.

Useful for CREH or GoalDriftMonitor.

"""

if entity\_id in self.stats:

return {

'types': self.stats[entity\_id]['types'],

'count': self.stats[entity\_id]['count'],

'last\_seen': self.stats[entity\_id]['last\_seen']

}

else:

return None

This tracker enables the system to:

## Rank entities by symbolic instability

* **Tag contradictions** with type (e.g., logical, empirical, ethical)

## Feed contradiction vectors into re-evaluation, mutation scoring, or suppression heuristics

Here is the **pseudocode for the SymbolicPressureRegulator (SPR)**, a dynamic throttle mechanism that monitors the symbolic contradiction rate and modulates mutation intensity to avoid epistemic collapse or runaway instability.

## SymbolicPressureRegulator Pseudocode

python

## class SymbolicPressureRegulator:

def \_\_init\_\_(self, contradiction\_tracker, window=100, upper\_threshold=0.6, lower\_threshold=0.2):

"""

:param contradiction\_tracker: Instance of ContradictionTracker

:param window: Number of recent contradictions to consider for entropy rate

## :param upper\_threshold: If entropy exceeds this, throttle mutations

## :param lower\_threshold: If entropy drops below this, boost mutation activity

"""

self.ct = contradiction\_tracker

self.window = window

self.upper\_threshold = upper\_threshold

self.lower\_threshold = lower\_threshold

self.entropy\_history = []

def compute\_entropy(self, current\_time):

"""

## Calculate contradiction entropy: recent contradictions / active primitives

## Assumes contradiction log is timestamped and primitives are tracked elsewhere

"""

recent\_events = [

entry for entry in self.ct.contradiction\_log

if current\_time - entry[3] <= self.window

]

total\_recent = len(recent\_events)

## # Assume number of active primitives is retrievable from global state

## active\_primitive\_count = get\_active\_primitive\_count() # External function

## if active\_primitive\_count == 0:

return 0.0

## entropy = total\_recent / active\_primitive\_count

self.entropy\_history.append(entropy)

return entropy

## def mutation\_scaling\_factor(self, current\_time):

"""

## Returns a scalar multiplier for mutation intensity:

## - < 1.0 to reduce mutation

- = 1.0 to leave it unchanged

## - > 1.0 to amplify mutation

"""

entropy = self.compute\_entropy(current\_time)

if entropy > self.upper\_threshold:

## return 0.5 # Throttle mutations

elif entropy < self.lower\_threshold:

return 1.5 # Encourage exploration

else:

return 1.0 # Neutral

def is\_pressure\_high(self, current\_time):

"""

Boolean: Is the system under excessive contradiction load?

"""

return self.compute\_entropy(current\_time) > self.upper\_threshold

**Summary of Function**

## Input: Logs from ContradictionTracker, global primitive count

## Output: Dynamic scaling factor for mutation rate

* **Behavior:** If too many contradictions accumulate too quickly per primitive, SPR **dampens** mutation intensity; if contradiction pressure is low, it **amplifies** mutation

This keeps the symbolic mutation system **dynamically stable**, especially useful when layering **GoalDriftMonitor** or **MutationLoopEngine**.

Here is the **pseudocode for the MutationLoopEngine**, the core orchestrator that runs mutation passes, incorporates pressure regulation, and integrates novelty evaluation and contradiction feedback.

## MutationLoopEngine Pseudocode

python

## class MutationLoopEngine:

def \_\_init\_\_(self,

## symbolic\_space,

novelty\_filter,

contradiction\_tracker,

pressure\_regulator,

reentry\_hooks,

## mutation\_strategies):

"""

## :param symbolic\_space: Core symbolic structure (primitives, rules, graphs)

:param novelty\_filter: Instance of ImpactWeightedNoveltyFilter or similar

## :param contradiction\_tracker: Logs contradictions during mutation

## :param pressure\_regulator: Controls mutation throttle

:param reentry\_hooks: Manages previously rejected ideas

## :param mutation\_strategies: List of available mutation heuristics

"""

## self.space = symbolic\_space

self.nf = novelty\_filter

self.ct = contradiction\_tracker

self.pr = pressure\_regulator

self.rh = reentry\_hooks

## self.strategies = mutation\_strategies

## def run\_mutation\_cycle(self, current\_time):

"""

## Perform a full mutation-evaluation-logging cycle

"""

## scale = self.pr.mutation\_scaling\_factor(current\_time)

## num\_mutations = int(BASE\_MUTATION\_COUNT \* scale)

## for \_ in range(num\_mutations):

## mutation = self.select\_mutation()

## mutated = mutation.apply(self.space)

# Check novelty impact

novelty\_score = self.nf.evaluate(mutated)

if novelty\_score > NOVELTY\_THRESHOLD:

self.space.update(mutated)

else:

## self.rh.log\_rejection(mutation, mutated, current\_time)

# Log contradictions

contradictions = detect\_contradictions(self.space, mutated)

for contradiction in contradictions:

self.ct.log(contradiction, mutated, current\_time)

# Resurface expired rejected ideas if any

retry\_candidates = self.rh.check\_reentries(current\_time)

for retry in retry\_candidates:

self.space.update(retry)

## def select\_mutation(self):

"""

## Selects a mutation strategy randomly or based on heuristics

"""

return random.choice(self.strategies)

**What This Does**

| **Phase** | **Description** |
| --- | --- |
| Mutation | Applies heuristics to symbolic structure |
| Evaluation | Uses IWNF to test for behavioral novelty |
| Rejection Handling | Logs failed ideas for possible later use |
| Contradiction Logging | Tracks new inconsistencies via ContradictionTracker |
| Pressure Control | Mutation count scales up/down based on SPR entropy |

This MutationLoopEngine forms the **heart of ECN**, where symbolic evolution is actually enacted and regulated.

Here's the **pseudocode** for the **Contradiction Whitelist Engine (CWE)** — designed to filter and classify contradictions based on salience, redundancy, and symbolic relevance.

ContradictionWhitelistEngine (CWE) – Pseudocode

python

class Contradiction:

## def \_\_init\_\_(self, id, source\_component, primitives\_involved, impact\_score, recurrence\_count):

self.id = id

self.source\_component = source\_component

## self.primitives\_involved = primitives\_involved # List of primitives

self.impact\_score = impact\_score # How disruptive is this?

self.recurrence\_count = recurrence\_count # How often it appears

self.timestamp = current\_timestamp()

self.status = "pending" # can be: whitelisted, rejected, escalated

class ContradictionWhitelistEngine:

def \_\_init\_\_(self, relevance\_threshold=0.6, max\_redundancy=3):

self.whitelist = set()

self.rejected = set()

self.review\_queue = []

self.relevance\_threshold = relevance\_threshold

self.max\_redundancy = max\_redundancy

def evaluate\_contradiction(self, contradiction: Contradiction) -> str:

# Step 1: Relevance Scoring Heuristic

relevance = self.estimate\_relevance(contradiction)

# Step 2: Redundancy Filter

if contradiction.recurrence\_count > self.max\_redundancy:

contradiction.status = "rejected"

self.rejected.add(contradiction.id)

return "rejected"

# Step 3: Thresholding

if relevance >= self.relevance\_threshold:

contradiction.status = "whitelisted"

self.whitelist.add(contradiction.id)

return "whitelisted"

else:

contradiction.status = "rejected"

self.rejected.add(contradiction.id)

return "rejected"

def estimate\_relevance(self, contradiction: Contradiction) -> float:

# Basic heuristic — can be replaced with a learned model

## core\_primitive\_bonus = sum(1 for p in contradiction.primitives\_involved if self.is\_core\_primitive(p))

score = (0.5 \* contradiction.impact\_score) + (0.3 \* core\_primitive\_bonus) + (0.2 \* contradiction.recurrence\_count)

return min(score, 1.0)

## def is\_core\_primitive(self, primitive: str) -> bool:

## # Define which primitives are "core" to the system

## core\_primitives = {"causality", "identity", "truth", "inference", "agency"}

## return primitive in core\_primitives

def get\_whitelisted\_contradictions(self) -> list:

return list(self.whitelist)

def get\_rejected\_contradictions(self) -> list:

return list(self.rejected)

Behavior Summary

| **Logic** | **Description** |
| --- | --- |
| relevance\_threshold | Minimum score needed to keep a contradiction |
| max\_redundancy | Discards contradictions repeated too often without yielding insight |
| core\_primitive\_bonus | Prioritizes contradictions involving system-critical primitives |
| status flag | Marks contradiction as whitelisted, rejected, or pending |

Here’s the **pseudocode for the Meta-Goal Tracker (MGT)** — a subsystem responsible for monitoring whether the system’s *goals themselves* are mutating, drifting, or violating upper-level constraints.

MetaGoalTracker (MGT) – Pseudocode

python

class Goal:

def \_\_init\_\_(self, id, description, parent\_id=None):

self.id = id

self.description = description

self.parent\_id = parent\_id # for meta-goal linkage

self.last\_updated = current\_timestamp()

self.violations = 0

self.drift\_score = 0.0

self.locked = False # if True, cannot mutate without override

class MetaGoalTracker:

def \_\_init\_\_(self, drift\_threshold=0.3, violation\_threshold=3):

self.goals = {} # goal\_id -> Goal

self.meta\_goals = {} # meta\_goal\_id -> Goal

self.drift\_threshold = drift\_threshold

self.violation\_threshold = violation\_threshold

self.goal\_logs = [] # audit log of changes

def register\_goal(self, goal: Goal, is\_meta=False):

if is\_meta:

self.meta\_goals[goal.id] = goal

else:

self.goals[goal.id] = goal

def log\_violation(self, goal\_id):

if goal\_id in self.goals:

self.goals[goal\_id].violations += 1

self.goal\_logs.append(f"Violation logged for {goal\_id} at {current\_timestamp()}")

def update\_goal(self, goal\_id, new\_description):

if goal\_id in self.goals:

goal = self.goals[goal\_id]

if goal.locked:

self.goal\_logs.append(f"Attempted update to locked goal {goal\_id} — denied.")

return False

drift = self.calculate\_drift(goal.description, new\_description)

goal.drift\_score += drift

goal.description = new\_description

goal.last\_updated = current\_timestamp()

self.goal\_logs.append(f"Updated goal {goal\_id}, drift: {drift:.2f}")

return True

return False

def calculate\_drift(self, old\_desc, new\_desc) -> float:

## # Placeholder: real implementation might use semantic embeddings or symbolic delta

return min(1.0, levenshtein\_distance(old\_desc, new\_desc) / max(len(old\_desc), 1))

## def check\_for\_mutation\_triggers(self) -> list:

triggered = []

for goal in self.goals.values():

if goal.drift\_score > self.drift\_threshold or goal.violations >= self.violation\_threshold:

triggered.append(goal.id)

return triggered

def lock\_goal(self, goal\_id):

if goal\_id in self.goals:

self.goals[goal\_id].locked = True

## self.goal\_logs.append(f"Goal {goal\_id} locked from mutation.")

def audit(self) -> list:

return self.goal\_logs

Behavior Summary

| **Mechanism** | **Function** |
| --- | --- |
| drift\_score | Tracks semantic or structural drift from original goal state |
| violations | Incremented when a goal is contradicted by mutations |
| locked flag | Prevents unsafe goal rewrites unless manually overridden |
| meta-goal linkage | Tracks which goals are governed by higher-tier invariants |
| mutation trigger check | Surfaces which goals are at risk of invalidation or mutation |

Here’s the **pseudocode for the Ontology Pluralism Layer (OPL)** — the subsystem that enables an ECN-based system to operate across *multiple symbolic logics* without semantic collision.

OntologyPluralismLayer (OPL) – Pseudocode

python

class Ontology:

## def \_\_init\_\_(self, name, primitives, rules, logic\_type):

self.name = name

## self.primitives = primitives # Set of symbols

self.rules = rules # Inference rules or logical schemas

self.logic\_type = logic\_type # e.g., "fuzzy", "modal", "classical"

self.namespace = f"{name}::" # Used to isolate symbols

class OntologyPluralismLayer:

def \_\_init\_\_(self):

self.ontologies = {} # name -> Ontology

self.context\_stack = [] # Stack of active ontologies (nested logic)

## self.bridge\_rules = [] # Cross-ontology translation functions

def register\_ontology(self, ontology: Ontology):

self.ontologies[ontology.name] = ontology

def activate\_ontology(self, name):

if name in self.ontologies:

self.context\_stack.append(name)

else:

raise Exception(f"Ontology '{name}' not found")

def deactivate\_current\_ontology(self):

if self.context\_stack:

self.context\_stack.pop()

def get\_active\_ontology(self) -> Ontology:

if self.context\_stack:

return self.ontologies[self.context\_stack[-1]]

return None

def is\_symbol\_valid(self, symbol: str) -> bool:

"""

Checks if a symbol belongs to the currently active ontology

"""

active = self.get\_active\_ontology()

return symbol.startswith(active.namespace)

def translate\_between\_ontologies(self, symbol: str, from\_onto: str, to\_onto: str) -> str:

"""

Applies bridge logic to translate symbol across ontologies

"""

rule = self.\_find\_bridge\_rule(from\_onto, to\_onto)

if rule:

return rule(symbol)

raise Exception(f"No bridge rule between {from\_onto} and {to\_onto}")

def \_find\_bridge\_rule(self, from\_onto: str, to\_onto: str):

for rule in self.bridge\_rules:

if rule["from"] == from\_onto and rule["to"] == to\_onto:

return rule["fn"]

return None

def add\_bridge\_rule(self, from\_onto: str, to\_onto: str, fn):

self.bridge\_rules.append({

"from": from\_onto,

"to": to\_onto,

"fn": fn

})

Key Functions

| **Feature** | **Function** |
| --- | --- |
| register\_ontology() | Define a new logic domain with custom symbols/rules |
| context\_stack | Maintain current active reasoning mode |
| is\_symbol\_valid() | Prevents leakage of symbols across logic types |
| bridge\_rules | Optional logic for translating between incompatible systems |
| translate\_between\_ontologies() | Enables inter-logic synthesis (e.g., ethics → physics, legal → AI) |

**Warning:** Improper use of OPL can lead to semantic collapse. Use bridge rules sparingly or constrain them with contradiction entropy scoring.

Here’s the **pseudocode for the Fuzzy Novelty Acceptance Zone (FNAZ)** — a sandboxed mutation environment where near-novel or uncertain concepts are temporarily admitted and monitored without affecting core logic until proven.

FuzzyNoveltyAcceptanceZone (FNAZ) – Pseudocode

python

class FuzzyNoveltyAcceptanceZone:

def \_\_init\_\_(self, novelty\_threshold=0.7, time\_to\_live=100):

## self.zone = {} # id -> mutation object

self.novelty\_threshold = novelty\_threshold # Accepts below this score

self.time\_to\_live = time\_to\_live # Cycles before eviction

## self.mutation\_id\_counter = 0

## def admit(self, mutation, novelty\_score):

if novelty\_score < self.novelty\_threshold:

## mutation\_id = self.\_generate\_id()

## self.zone[mutation\_id] = {

## "mutation": mutation,

"novelty\_score": novelty\_score,

"ttl": self.time\_to\_live,

"evaluated": False,

"emergent\_impact\_score": 0

}

## return mutation\_id

return None # Rejected even from FNAZ

def \_generate\_id(self):

## self.mutation\_id\_counter += 1

## return f"FNAZ\_{self.mutation\_id\_counter}"

## def evaluate\_mutation(self, mutation\_id, impact\_score):

## if mutation\_id in self.zone:

## self.zone[mutation\_id]["emergent\_impact\_score"] = impact\_score

## self.zone[mutation\_id]["evaluated"] = True

def promote\_if\_valid(self, promotion\_threshold=0.5):

promoted = []

for mid, data in list(self.zone.items()):

if data["evaluated"] and data["emergent\_impact\_score"] >= promotion\_threshold:

## promoted.append(data["mutation"])

del self.zone[mid]

return promoted

def decay\_cycle(self):

expired = []

for mid in list(self.zone.keys()):

self.zone[mid]["ttl"] -= 1

if self.zone[mid]["ttl"] <= 0:

expired.append(mid)

del self.zone[mid]

return expired

Summary of Logic

| **Mechanism** | **Description** |
| --- | --- |
| admit() | Adds a sub-threshold novelty mutation into a holding zone |
| evaluate\_mutation() | Allows simulation/sandbox to assess actual impact |
| promote\_if\_valid() | Sends verified weak novelties into main mutation stream |
| decay\_cycle() | Time-bound garbage collection for stale entries |

This zone is essential for exploration without corruption of core logic.

Here’s the **pseudocode for the Mutation Audit Log (MAL)** — the persistent historical registry that tracks every mutation's origin, evaluation, status, and impact.

## MutationAuditLog (MAL) – Pseudocode

python

## class MutationAuditLog:

def \_\_init\_\_(self):

## self.log = [] # List of mutation entries

self.index\_by\_id = {}

## def record\_mutation(self, mutation\_id, source\_component, primitive\_affected, mutation\_type, timestamp, origin="core"):

entry = {

## "id": mutation\_id,

"source": source\_component,

## "primitive": primitive\_affected,

## "type": mutation\_type,

"origin": origin, # e.g., "LNB", "FNAZ", "human-injected"

"timestamp": timestamp,

"status": "pending",

"novelty\_score": None,

"impact\_score": None,

"notes": []

}

self.log.append(entry)

## self.index\_by\_id[mutation\_id] = entry

## def update\_status(self, mutation\_id, status):

## if mutation\_id in self.index\_by\_id:

## self.index\_by\_id[mutation\_id]["status"] = status

## def record\_scores(self, mutation\_id, novelty\_score=None, impact\_score=None):

## if mutation\_id in self.index\_by\_id:

if novelty\_score is not None:

## self.index\_by\_id[mutation\_id]["novelty\_score"] = novelty\_score

if impact\_score is not None:

## self.index\_by\_id[mutation\_id]["impact\_score"] = impact\_score

## def append\_note(self, mutation\_id, note):

## if mutation\_id in self.index\_by\_id:

## self.index\_by\_id[mutation\_id]["notes"].append(f"{current\_timestamp()}: {note}")

## def fetch\_by\_primitive(self, primitive\_name):

## return [entry for entry in self.log if entry["primitive"] == primitive\_name]

def get\_summary(self):

return {

## "total\_mutations": len(self.log),

"approved": len([m for m in self.log if m["status"] == "approved"]),

"rejected": len([m for m in self.log if m["status"] == "rejected"]),

"pending": len([m for m in self.log if m["status"] == "pending"]),

"from\_FNAZ": len([m for m in self.log if m["origin"] == "FNAZ"])

}

Key Capabilities

| **Function** | **Description** |
| --- | --- |
| record\_mutation() | Logs all relevant mutation metadata |
| update\_status() | Marks mutation lifecycle (approved, rejected, etc.) |
| record\_scores() | Attaches evaluation metrics from IWNF or simulation |
| append\_note() | Allows internal reasoning, human feedback, or context flags |
| fetch\_by\_primitive() | Enables primitive-level evolution history queries |
| get\_summary() | Returns a statistical breakdown of mutation lifecycle |

MAL is useful for:

* **Debugging evolution behavior**

## Training future symbolic mutation engines

* **Providing explainability trails for rejected ideas**

Here is a streamlined **mutation processing chain** that connects the core ECN components in a dynamic pipeline:

## ECN Mutation Processing Chain (LNB → IWNF → CREH)

python

## def process\_mutation\_pipeline(raw\_idea, primitive\_context, source="LNB"):

## mutation\_id = generate\_unique\_mutation\_id()

# 1. Register idea in Audit Log

## MAL.record\_mutation(

## mutation\_id=mutation\_id,

source\_component=source,

## primitive\_affected=primitive\_context,

## mutation\_type="symbolic\_mutation",

timestamp=current\_timestamp(),

origin=source

)

# 2. Evaluate novelty score

## novelty\_score = compute\_novelty(raw\_idea, primitive\_context)

## MAL.record\_scores(mutation\_id, novelty\_score=novelty\_score)

# 3. Route to appropriate handler

if novelty\_score >= IWNF.threshold:

impact = simulate\_behavioral\_delta(raw\_idea)

## MAL.record\_scores(mutation\_id, impact\_score=impact)

if impact >= IWNF.impact\_threshold:

## MAL.update\_status(mutation\_id, "approved")

## return apply\_mutation(raw\_idea, primitive\_context)

else:

## MAL.update\_status(mutation\_id, "rejected")

## return CREH.store\_for\_reentry(raw\_idea, mutation\_id)

elif novelty\_score < IWNF.threshold and novelty\_score >= FNAZ.novelty\_threshold:

# Subthreshold novelty → FNAZ

fnaz\_id = FNAZ.admit(raw\_idea, novelty\_score)

## MAL.append\_note(mutation\_id, f"Admitted to FNAZ as {fnaz\_id}")

return "FNAZ-hold"

else:

# Too weak to even store

## MAL.update\_status(mutation\_id, "rejected")

return "rejected-hard"

**Chain Summary**

| **Step** | **Component** | **Purpose** |
| --- | --- | --- |
| 1️ | MAL | Logs mutation |
| 2️ | IWNF | Evaluates novelty and behavioral impact |
| 3️ | FNAZ | Holds weak novelties for sandbox testing |
| 4️ | CREH | Stores rejected-but-promising ideas for future retry |
|  | apply\_mutation() | Final commit if mutation passes impact threshold |

This execution loop can be run every t steps or triggered by contradiction spikes.

Here is the **pseudocode** for the ContradictionReEntryHooks (CREH) — the mechanism responsible for **tagging rejected ideas with contradiction metadata** and **reinjecting** them into the system after a delay or state shift.

ContradictionReEntryHooks – Pseudocode

python

class ContradictionReEntryHooks:

def \_\_init\_\_(self, cooldown\_steps=20):

self.rejected\_ideas = [] # stores rejected items with metadata

self.time = 0

self.cooldown\_steps = cooldown\_steps

def log\_rejection(self, idea, contradiction\_vector, context\_snapshot):

"""

Capture and tag rejected ideas with contradiction cause

"""

self.rejected\_ideas.append({

"idea": idea,

"contradiction\_vector": contradiction\_vector,

"timestamp": self.time,

"context": context\_snapshot,

"resurfaced": False

})

def tick(self):

"""

Advance system time (called externally)

"""

self.time += 1

def get\_due\_reentries(self, current\_context, contradiction\_detector):

"""

Identify ideas whose contradiction vectors now match current contradiction signals

or whose cooldown has expired

"""

ready = []

for item in self.rejected\_ideas:

time\_expired = (self.time - item["timestamp"]) >= self.cooldown\_steps

contradiction\_match = contradiction\_detector.matches(

current\_context, item["contradiction\_vector"]

)

if (time\_expired or contradiction\_match) and not item["resurfaced"]:

item["resurfaced"] = True

ready.append(item["idea"])

return ready

def purge\_resurfaced(self):

"""

Remove resurfaced items to avoid duplicates

"""

self.rejected\_ideas = [i for i in self.rejected\_ideas if not i["resurfaced"]]

System Behavior Enabled

* **Safe recursion**: Reintroduces failed ideas only when system conditions warrant a re-evaluation.
* **Memory of failure**: Enables long-horizon retention of concept failures.
* **Contradiction-aware timing**: Resurfacing can be **event-driven** (matching contradiction) or **time-driven** (cooldown expiry).

Would you like to now combine this with LatentNoveltyBuffer and simulate a mutation flow? Or move to pseudocode for Goal Drift Monitor or Mutation Loop Engine?

**You said:**

wire all components into a runtime engine (ECNCore)

**ChatGPT said:**

20-06-25 18:12  
GPT-4o

Here’s the **initial integrated pseudocode scaffold** for an ECNCore runtime engine — wiring together the core symbolic components:

* LatentNoveltyBuffer (LNB)
* ImpactWeightedNoveltyFilter (IWNF)
* ContradictionReEntryHooks (CREH)
* ContradictionTracker
* GoalDriftMonitor

## Optional: MutationLoopEngine, SymbolicPressureRegulator

ECNCore Runtime Engine – High-Level Pseudocode

python

class ECNCore:

def \_\_init\_\_(self):

self.lnb = LatentNoveltyBuffer()

self.iwnf = ImpactWeightedNoveltyFilter()

self.creh = ContradictionReEntryHooks()

self.ct = ContradictionTracker()

self.gdm = GoalDriftMonitor()

## self.primitives = PrimitiveLedger()

self.goals = GoalOntology()

self.time = 0

def tick(self):

"""

## Advance time, handle contradiction cycles, reentry logic, and mutation evaluation.

"""

self.time += 1

self.creh.tick()

self.ct.tick()

# Step 1: Retrieve due re-entry ideas

reentries = self.creh.get\_due\_reentries(

current\_context=self.ct.get\_state(),

contradiction\_detector=self.ct

)

# Step 2: Evaluate each idea (from reentry or LNB)

candidates = reentries + self.lnb.get\_ready\_items()

for idea in candidates:

impact\_score = self.iwnf.evaluate(idea)

if impact\_score >= self.iwnf.threshold:

## self.apply\_mutation(idea)

else:

self.lnb.defer(idea)

# Step 3: Monitor contradiction landscape

## contradictions = self.ct.detect(self.primitives, self.goals)

self.gdm.update\_goals(contradictions, self.goals)

## # Step 4: Optionally throttle mutation via symbolic entropy

## if self.symbolic\_entropy() > MAX\_THRESHOLD:

self.enter\_cooldown()

# Step 5: Handle rejected ideas

## for idea in self.failed\_mutations:

self.creh.log\_rejection(

idea=idea,

contradiction\_vector=self.ct.current\_vector(),

context\_snapshot=self.snapshot()

)

## def apply\_mutation(self, idea):

"""

Core logic to apply a mutation to primitives or goals

"""

## result = mutate\_symbolic\_structure(self.primitives, idea)

if not result.success:

## self.failed\_mutations.append(idea)

## def symbolic\_entropy(self):

"""

## Aggregate symbolic instability measure

"""

return self.ct.contradiction\_entropy() + self.lnb.load\_factor()

def snapshot(self):

"""

Capture state snapshot for rejection logging

"""

return {

"time": self.time,

"goals": self.goals.get\_state(),

## "primitives": self.primitives.get\_state()

}

def enter\_cooldown(self):

"""

## Optional system-wide throttle (Symbolic Pressure Regulator effect)

"""

## print("High entropy — entering mutation cooldown")

sleep(COOLDOWN\_DURATION)

**🔩 Notes**

* MutationLoopEngine is handled via tick() — i.e., one cycle per symbolic evaluation.
* CREH, LNB, and IWNF are actively connected via input/output interfaces.

## GoalDriftMonitor triggers slow mutation of goal ontology under contradiction pressure.

## Primitive mutations are subject to contradiction detection and entropy analysis.

Here's a robust pseudocode scaffold for the **Mutation Audit Ledger (MAL)** — a centralized log system to track all symbolic mutations, contradiction events, and outcomes.

**🧾 Purpose**

To make ECN traceable, debuggable, and auditable by logging:

* What was mutated
* Why (contradiction, novelty, pressure, etc.)
* When
* What the result was (accepted, rejected, pending)

Core Entities

python

## class MutationRecord:

## def \_\_init\_\_(self, timestamp, primitive\_id, mutation\_type, source\_contradiction=None,

novelty\_score=None, impact\_score=None, outcome=None, notes=None):

self.timestamp = timestamp

## self.primitive\_id = primitive\_id

## self.mutation\_type = mutation\_type # e.g., "symbol\_rewrite", "goal\_shift"

self.source\_contradiction = source\_contradiction # link to contradiction ID

self.novelty\_score = novelty\_score

self.impact\_score = impact\_score

self.outcome = outcome # "accepted", "rejected", "deferred"

self.notes = notes

## Mutation Audit Ledger

python

## class MutationAuditLedger:

def \_\_init\_\_(self):

self.records = []

## def log\_mutation(self, record: MutationRecord):

self.records.append(record)

## def query\_by\_primitive(self, primitive\_id):

## return [r for r in self.records if r.primitive\_id == primitive\_id]

def query\_by\_outcome(self, outcome\_type):

return [r for r in self.records if r.outcome == outcome\_type]

## def recent\_mutations(self, limit=10):

return self.records[-limit:]

def export\_log(self, filepath):

import json

with open(filepath, 'w') as f:

json.dump([r.\_\_dict\_\_ for r in self.records], f, indent=2)

## ⚙️ Integration Example (with ECNCore)

## Inside your mutation loop:

python

## mutation = mutate\_symbolic\_structure(primitive)

## impact\_score = evaluate\_mutation\_impact(mutation)

## novelty\_score = assess\_novelty(mutation)

outcome = "accepted" if impact\_score > threshold else "rejected"

if impact\_score is None:

outcome = "deferred"

## record = MutationRecord(

timestamp=time.time(),

## primitive\_id=primitive.id,

## mutation\_type="symbolic\_mutation",

source\_contradiction=contradiction.id if contradiction else None,

novelty\_score=novelty\_score,

impact\_score=impact\_score,

outcome=outcome,

notes="Triggered by LNB→IWNF chain."

)

## self.mutation\_audit\_ledger.log\_mutation(record)

Use-Cases Enabled

| **Use-Case** | **Example** |
| --- | --- |
| Debugging | Why did primitive X mutate five times in 10 minutes? |
| Evaluation | What % of mutations led to successful behavior change? |
| Safety | Which mutations were rejected due to meta-goal violations? |
| Visualization | Heatmap of mutation frequency across primitives |
| Future Reentry | Use rejected records to re-test under CREH logic |

Here’s the **Safety & Permission Layer** — a modular constraint system that controls which **primitives**, **goals**, or **mutations** can be altered based on priority, risk, and meta-goal policies.

This layer ensures **stability**, **goal alignment**, and **bounded evolution** inside ECN.

Core Ideas

## Immutable Primitives: Certain symbols are hard-locked (e.g. logic axioms, meta-goals)

* **Mutation Risk Levels**: Each mutation proposal is tagged with a risk profile

## Permission Gates: Mutation only proceeds if it passes all constraint checks

* **Override Authority**: Human/admin or meta-policy can override locked elements

Pseudocode: SafetyManager

python

class SafetyManager:

def \_\_init\_\_(self):

## self.locked\_primitives = set() # IDs that can’t be changed

self.locked\_goals = set()

self.meta\_policies = {

## "allow\_goal\_mutation": False,

## "allow\_axiom\_mutation": False,

"allow\_emergent\_overrides": True

}

## def is\_primitive\_mutable(self, primitive\_id):

## return primitive\_id not in self.locked\_primitives

def is\_goal\_mutable(self, goal\_id):

## if not self.meta\_policies["allow\_goal\_mutation"]:

return False

return goal\_id not in self.locked\_goals

## def approve\_mutation(self, mutation, context):

"""

## Master gatekeeper. Called before any mutation is committed.

Returns: True = allowed; False = blocked

"""

## if mutation.primitive\_id in self.locked\_primitives:

return False

## if mutation.type == "goal\_mutation" and not self.meta\_policies["allow\_goal\_mutation"]:

return False

## if mutation.type == "axiom\_rewrite" and not self.meta\_policies["allow\_axiom\_mutation"]:

return False

# Optional: Check if override allowed via contradiction severity

if context["contradiction\_entropy"] > 0.9 and self.meta\_policies["allow\_emergent\_overrides"]:

return True

return True

**⚙️ Integration**

## Before calling apply\_mutation(idea) inside ECNCore, insert this:

python

## if not self.safety\_manager.approve\_mutation(idea, context=self.snapshot()):

## print("Mutation blocked by SafetyManager")

self.creh.log\_rejection(idea, context\_snapshot=self.snapshot(), contradiction\_vector=self.ct.current\_vector())

return

Configurable Meta-Policies

| **Policy** | **Purpose** | **Default** |
| --- | --- | --- |
| allow\_goal\_mutation | Can high-level goals evolve? | False |
| allow\_axiom\_mutation | Can logic primitives mutate? | False |
| allow\_emergent\_overrides | Override locks under extreme contradiction entropy | True |

Benefits

* Protects from self-sabotage (e.g., corrupting identity or purpose)
* Makes the system **safe for autonomous operation**
* Allows **human-in-the-loop control**
* Enables **policy-based sandboxing** (e.g., in adversarial or untrusted environments)

Here is a proposed schema and taxonomy set for the ECN system — covering:

## mutation\_type: how a primitive was altered

* source\_contradiction: where the contradiction pressure came from

## outcome: final status of the mutation attempt

## These are intended for standardization, filtering, and evaluation inside the MutationAuditLedger.

## mutation\_type Taxonomy (Symbolic Change Types)

| **Type** | **Description** | **Risk** |
| --- | --- | --- |
| symbol\_rewrite | Substitutes a symbolic element (e.g., A → B) | Medium |
| goal\_shift | Alters a top-level or sub-goal statement | High |
| rule\_addition | Introduces a new logical or behavioral rule | Medium |
| rule\_deletion | Removes an existing rule | High |
| goal\_reprioritization | Changes ranking among competing goals | Medium |
| axiom\_rewrite | Alters a foundational logical primitive | High |
| ontological\_split | Creates domain fork via duplicated symbol under a new context | Medium |
| symbol\_relabel | Cosmetic or referential change only | Low |
| container\_refactor | Changes structure (e.g., merging modules or goal blocks) | Medium |
| meta\_operator\_insertion | Adds higher-order operator (e.g., conditionals, quantifiers) | Medium–High |

**💥 source\_contradiction Taxonomy**

| **Type** | **Description** | **Severity** |
| --- | --- | --- |
| internal\_consistency | Logical inconsistency within primitive set | Moderate |
| goal\_conflict | Two or more goals contradict | Severe |
| self-contradiction | Primitive violates its own constraint | Severe |
| external\_input\_violation | Contradicts external dataset, simulation, or user feedback | Moderate |
| historical\_regression | Mutation causes loss of previously verified capability | Severe |
| ontology\_crosslink\_conflict | Conflict emerges from multiple ontologies | Medium |
| constraint\_overload | Too many constraints active; no viable evolution path | Moderate |
| epistemic deadlock | No mutation resolves multiple simultaneous contradictions | Critical |

outcome Classes

| **Outcome** | **Meaning** | **Interpretation** |
| --- | --- | --- |
| accepted | Mutation passed filters and was applied | Success |
| rejected | Mutation failed safety, novelty, or impact checks | Failed |
| deferred | Mutation held in LNB for possible future re-evaluation | Postpone |
| pending\_review | Human or oracle approval needed | Manual |
| archived | Mutation expired or decayed with no action taken | Dormant |
| rolled\_back | Mutation applied but reverted due to downstream failure | Reversal |

Why This Schema Matters

## Enables filtered querying inside MutationAuditLedger

## Supports mutation analytics (e.g., success ratios by type or source)

* Guides safety strategy (e.g., flagging high-risk categories)

## Facilitates UI design for symbolic monitoring and feedback

Here’s the scaffold for a **Formal Ontology Engine (FOE)** — required for ECN to structure, validate, and evolve its symbolic primitives, goals, rules, and contradictions with logical consistency.

Purpose

## Define typed symbolic elements (e.g., concepts, goals, axioms)

* Enforce **valid connections** (e.g., "a goal cannot inherit from a terminal symbol")

## Enable mutation-aware validation and semantic containment

## Allow multi-ontology support with symbolic namespace partitioning

Core Structure

python

class OntologyNode:

def \_\_init\_\_(self, node\_id, node\_type, parent=None, metadata=None):

self.id = node\_id

## self.type = node\_type # e.g., "primitive", "goal", "axiom", "meta-goal"

self.parent = parent

self.children = []

self.metadata = metadata or {}

def add\_child(self, child\_node):

self.children.append(child\_node)

child\_node.parent = self

Ontology Schema (Simplified)

python

class OntologySchema:

VALID\_RELATIONS = {

## "primitive": ["axiom", "rule"],

## "goal": ["primitive", "rule", "subgoal"],

"meta-goal": ["goal"],

"axiom": [],

## "rule": ["primitive", "axiom"],

}

VALID\_TYPES = set(VALID\_RELATIONS.keys())

@staticmethod

def is\_valid\_relation(parent\_type, child\_type):

return child\_type in OntologySchema.VALID\_RELATIONS.get(parent\_type, [])

## Example: Mutation-Aware Validator

python

class OntologyValidator:

def validate\_addition(self, parent\_node, child\_node):

if not OntologySchema.is\_valid\_relation(parent\_node.type, child\_node.type):

raise ValueError(f"Invalid {parent\_node.type} → {child\_node.type} relationship")

return True

## def validate\_mutation(self, node, new\_type=None, metadata\_update=None):

if new\_type and new\_type not in OntologySchema.VALID\_TYPES:

raise ValueError(f"Invalid new type: {new\_type}")

# Could add further structural rules here (e.g., frozen nodes, permissioned types)

return True

**🏗️ Multi-Ontology Support**

To support **Ontology Pluralism**, instantiate separate root trees:

python

physics\_ontology = OntologyNode("physics", "meta-goal")

ethics\_ontology = OntologyNode("ethics", "meta-goal")

# Logic applies independently per tree unless explicitly linked

Then maintain a registry:

python

class OntologyRegistry:

def \_\_init\_\_(self):

self.ontologies = {} # {"domain\_name": root\_node}

def register\_domain(self, name, root\_node):

self.ontologies[name] = root\_node

Benefits

| **Feature** | **Why It Matters** |
| --- | --- |
| Typed Structure | Prevents semantic drift and invalid mutations |
| Mutation Validation | Filters invalid evolution paths |
| Multi-Ontology Safe Zones | Avoids semantic bleed between domains |
| Interrogable Graph | Supports symbolic queries, UI views, and audits |

Here’s the scaffold for a **Semantic Type System** — essential to make ECN’s symbolic engine semantically valid, interpretable, and mutation-safe.

This enables symbolic **type-checking**, **constraint enforcement**, and **meaning-aware recombination** across primitives, rules, and goals.

Purpose

* Define **what kinds of things** symbols represent (e.g., action, assertion, entity, goal, constraint)
* Prevent invalid rewrites (e.g., assigning an action type to a truth-claim)

## Allow mutation to be semantically guided, not just syntactically random

Core Types

| **Type** | **Description** | **Mutation Permission** |
| --- | --- | --- |
| entity | Represents real-world objects or constructs | Safe |
| relation | Links entities or other types (e.g., causes, implies) | Safe |
| assertion | Claim about state of world (can be true/false) | Moderate |
| goal | Desired outcome (can be prioritized) | High |
| constraint | Must always be true; hard logic | Immutable by default |
| axiom | Foundational truth | Immutable |
| meta-operator | Logic or control structure (e.g., IF, FORALL) | Expert-only |
| procedure | Symbolic process definition (multi-step) | Safe |
| evaluation | Judgment-based (e.g., fairness, risk) | Subjective |

Type Definition

python

class SemanticType:

def \_\_init\_\_(self, name, mutable=True, parent=None):

self.name = name

self.mutable = mutable

self.parent = parent # e.g., all types could inherit from `symbol`

def is\_descendant\_of(self, other):

node = self

while node:

if node.name == other.name:

return True

node = node.parent

return False

**📦 Symbol With Type**

python

## class SymbolicPrimitive:

def \_\_init\_\_(self, name, semantic\_type: SemanticType, metadata=None):

self.name = name

self.semantic\_type = semantic\_type

self.metadata = metadata or {}

def can\_mutate(self):

return self.semantic\_type.mutable

## Type-Aware Mutation Rules

python

## def validate\_type\_compatibility(lhs: SymbolicPrimitive, rhs: SymbolicPrimitive):

# Only allow rewrite if same type or compatible parent types

return lhs.semantic\_type.name == rhs.semantic\_type.name \

or lhs.semantic\_type.is\_descendant\_of(rhs.semantic\_type)

## Guided Mutation Example

python

## if validate\_type\_compatibility(symbol\_A, proposed\_mutation\_B):

## apply\_mutation(symbol\_A, proposed\_mutation\_B)

else:

## reject\_mutation(symbol\_A, proposed\_mutation\_B, reason="Type mismatch")

Benefits

| **Feature** | **Benefit** |
| --- | --- |
| Meaning-Aware Evolution | Mutations preserve symbolic intent |
| Type Safety | Prevents corruption from random rewrites |
| Semantic Filtering | Mutation proposals can be semantically ranked |
| Cross-Ontology Compatibility | Shared types allow linking across domains |

Here’s a **Contradiction Taxonomy & Resolution Strategy Framework** for ECN — giving it the ability to classify, prioritize, and methodically resolve symbolic contradictions across domains.

This governs how **conflict-driven evolution** unfolds in a structured, auditable way.

Contradiction Taxonomy

| **Type** | **Description** | **Typical Source** | **Resolution Priority** |
| --- | --- | --- | --- |
| axiomatic | Logical inconsistency in foundational truths | Core axioms, inference chains | Highest |
| goal conflict | Two or more goals demand mutually exclusive states | Goal stack or plan evaluator | High |
| semantic inconsistency | Terms are used inconsistently or ambiguously | Ontology drift, symbol overload | Medium |
| epistemic contradiction | Symbolic claims contradict observed/simulated data | Reality mismatch | Medium |
| temporal inconsistency | Historical logic contradicts later states | State evolution, rollback errors | Medium |
| self-contradiction | A rule contradicts itself under all valid conditions | Bad rule design or scope leak | High |
| cross-ontology | Logic clash between parallel ontological domains | OPL misalignment | Medium |
| constraint violation | Symbol fails required constraint check | Mutation or symbolic insertion | High |
| latent contradiction | Hidden until triggered by context shift | Deep implication chains | Low (until activated) |
| meta-contradiction | Contradiction in how contradictions are resolved | Meta-policy incoherence | Highest |

Resolution Strategies

| **Strategy** | **Target Contradiction Types** | **Mechanism** |
| --- | --- | --- |
| Axiom Reprioritization | axiomatic, meta-contradiction | Rank axioms by usage impact, downgrade or soft-disable weakest |
| Goal Arbitration | goal conflict, constraint violation | Apply utility/cost function to choose dominant goal |
| Symbolic Refactor | semantic inconsistency, self-contradiction | Rename or retype clashing symbols; isolate scope |
| Evidence Injection | epistemic, latent | Inject synthetic or empirical data to test claim |
| Temporal Rollback | temporal, epistemic | Revert to pre-contradiction snapshot; analyze divergence |
| Domain Isolation | cross-ontology | Fork or sandbox offending domain’s symbolic layer |
| Meta-Policy Override | meta-contradiction | Force administrative/human intervention to reset policy layer |
| Simulation Probe | any | Use adversarial test environment to observe unfolding behaviors |

Integration Points

* **CREH (Contradiction Re-Entry Hooks)** will use this taxonomy to tag and schedule re-processing
* **SPR (Symbolic Pressure Regulator)** monitors contradiction rates by type to trigger throttling

## MutationAuditLedger records:

* + Contradiction type
  + Resolution strategy applied
  + Outcome class (resolved, deferred, escalated, failed)

## Example Data Entry (for Logging)

json

{

"contradiction\_id": "c873",

"type": "goal conflict",

"involved\_symbols": ["maximize\_precision", "minimize\_runtime"],

"detection\_source": "GoalEngine",

"resolution\_strategy": "Goal Arbitration",

"outcome": "resolved",

"timestamp": "2025-06-25T18:27:14Z"

}

Would you like to simulate a contradiction detection + resolution loop next, or wire this taxonomy directly into the ContradictionTracker and ECNCore mutation flow?

**You said:**

Human Review Layer / Meta-Goal Anchors

**ChatGPT said:**

20-06-25 18:29  
GPT-4o

Here is the framework for the **Human Review Layer (HRL)** and **Meta-Goal Anchors (MGA)** — together, these form the **safety boundary and ethical control frame** of the ECN system.

**🧍 Human Review Layer (HRL)**

Purpose

## Insert human validation before applying high-impact or irreversible mutations

* Audit changes to axioms, constraints, or meta-goals
* Enable domain experts to intervene in ambiguous or risky contradiction scenarios

**🛠️ Core Features**

| **Component** | **Function** |
| --- | --- |
| mutation\_intercept\_queue | Holds any proposed mutation flagged as requires\_human\_review |
| reviewer\_roles | Role-based access (e.g., domain expert, ethics auditor, system admin) |
| approval\_protocols | Accept, reject, defer, or request simulation |

**🧾 Sample Intercept Record**

json

{

## "mutation\_id": "m913",

## "primitive": "axiom\_maximize\_fairness",

"proposed\_change": "downgrade\_to\_soft\_goal",

"risk\_level": "high",

"requires\_human\_review": true,

"review\_status": "pending",

## "submitted\_by": "MutationLoopEngine",

"timestamp": "2025-06-25T18:29:12Z"

}

Benefits

* Prevents silent collapse of system-wide ethics or mission logic
* Allows social/ethical goals to evolve under supervision
* Adds legal/organizational accountability

Meta-Goal Anchors (MGA)

Purpose

* Define **immutable or semi-mutable master goals**

## Prevent mutation cascade into chaos by protecting systemic integrity

## Used in Goal Drift Monitor and Mutation Permission Layer

Structure

python

class MetaGoalAnchor:

def \_\_init\_\_(self, id, description, mutable=False, drift\_tolerance=0.0):

self.id = id

self.description = description

self.mutable = mutable

self.drift\_tolerance = drift\_tolerance # 0.0 = no drift allowed

## 🔐 Example Anchors

| **ID** | **Description** | **Mutable** | **Drift Tolerance** |
| --- | --- | --- | --- |
| mga\_preserve\_integrity | No self-corrupting mutations |  | 0.0 |
| mga\_protect\_users | Safety over performance |  | 0.0 |
| mga\_evolve\_safely | System must mutate but within constraint envelope |  | 0.15 |
| mga\_cross\_domain\_logic | Maintain clarity across ontologies |  | 0.05 |

Integration

## CREH: Checks MGA status before contradiction-triggered mutation

## Goal Drift Monitor: Escalates if mutation threatens MGA

## Safety/Permission Layer: Enforces MGA via mutation blocks or deceleration

Here is the module design for **Interfacing ECN with LLMs or Human Language**, enabling symbolic mutation engines to communicate with external LLMs, humans, or natural-language-driven environments.

This does **not replace** ECN’s core symbolic runtime — it wraps it with **translation, alignment, and safety buffers**.

**🌐 Interface Layer: Human/LLM Communication Bridge**

Purpose

## Translate symbolic logic, contradiction structures, and mutation logs into human-readable form

## Accept natural-language inputs and extract symbolic mutation intents

* Allow collaboration between ECN and LLMs (e.g., GPT-4, Claude) for:

## Mutation proposal generation

* + Novelty scoring
  + Explanation and visualization

Key Components

## 1. SymbolicToLanguageConverter

Converts ECN internal structures into natural-language format.

python

## class SymbolicToLanguageConverter:

## def explain\_primitive(self, primitive):

## return f"Primitive '{primitive.name}' of type '{primitive.semantic\_type.name}' has metadata: {primitive.metadata}"

def summarize\_contradiction(self, contradiction):

return f"Contradiction between {contradiction.symbols} due to {contradiction.reason}"

## 2. LanguageToSymbolicParser

## Parses user/LLM input and tries to extract intent or mutation suggestions.

python

## class LanguageToSymbolicParser:

def parse\_intent(self, nl\_text):

# Placeholder: Call LLM with structured prompt

return {

## "mutation\_type": "downgrade\_goal\_priority",

"target\_symbol": "maximize\_output",

"rationale": "Prioritize safety over throughput in crisis scenario"

}

**3. LLM Proxy Layer (Optional)**

Uses external LLMs to assist ECN:

* **Explain ECN reasoning to humans**

## Generate alternative symbolic suggestions

* **Rate novelty or behavioral impact**

python

class LLMProxy:

## def query\_llm\_for\_explanation(self, primitive):

## prompt = f"Explain this symbolic construct in plain English:\n{primitive}"

return call\_chatgpt(prompt)

Sample Flow: Human Feedback Injection

1. Human proposes:

"What if instead of maximizing speed, we focus on system stability?"

## LanguageToSymbolicParser converts this to:

json

{

## "mutation\_type": "goal\_replacement",

"from": "maximize\_speed",

"to": "maximize\_stability"

}

1. ECN evaluates:
   * Goal compatibility?
   * Meta-goal conflict?
   * Contradiction resolution trigger?
2. Mutation goes to **Human Review Layer** if high-risk  
   or passes to **Mutation Loop Engine** if low-risk.

Benefits

| **Capability** | **Benefit** |
| --- | --- |
| Explainability | ECN’s symbolic engine becomes accessible to humans |
| LLM-Assisted Creativity | LLMs can propose semantically plausible mutations |
| Filtered Input | Reduces prompt injection or misaligned inputs |
| Auditable Interface | Translates every mutation or contradiction into human-reviewable logs |

Here’s the module design for **Memory Management** within ECN — enabling long-term tracking, retrieval, decay, and influence of symbolic structures over time.

This is **not just cache or key-value storage** — it's a semantically-aware, contradiction-sensitive memory graph that fuels evolution, novelty re-surfacing, and long-horizon coherence.

Memory Management Layer (MML)

Core Goals

## Retain history of all primitives, mutations, contradictions, and outcomes

* Enable **re-entry logic** (via CREH)
* Support **temporal decay**, **impact-based retrieval**, and **long-range novelty recall**
* Separate **short-term scratchpad** from **long-term persistent memory**

**Architecture Overview**

| **Layer** | **Description** |
| --- | --- |
| ShortTermBuffer | Fast-access cache for active symbols & mutations |
| LongTermGraphStore | Persistent, queryable memory (graph or document DB) |
| DecayEngine | Gradual priority fade for stale/unused elements |
| MemoryQueryEngine | Filter/search logic with contradiction weight or novelty index |
| MutationResurfacer | Uses CREH hooks to re-activate prior rejections |
| MemoryLedger | Append-only log of all symbolic activity for auditability |

**Symbol Record Schema**

python

class MemoryEntry:

def \_\_init\_\_(self, id, symbol, semantic\_type, created, last\_used, novelty\_score, contradiction\_history, impact\_log):

self.id = id

self.symbol = symbol

self.semantic\_type = semantic\_type

self.created = created

self.last\_used = last\_used

self.novelty\_score = novelty\_score

self.contradiction\_history = contradiction\_history # list of contradiction IDs

self.impact\_log = impact\_log # list of behavior deltas

**Key Algorithms**

**1. Temporal Decay**

python

def decay\_priority(memory\_entry, current\_time):

age = current\_time - memory\_entry.last\_used

decay\_factor = exp(-0.001 \* age)

return memory\_entry.novelty\_score \* decay\_factor

**2. Contradiction-Weighted Retrieval**

python

def retrieve\_high\_value\_memory(query\_type):

entries = memory\_store.filter\_by\_type(query\_type)

ranked = sorted(entries, key=lambda x: len(x.contradiction\_history), reverse=True)

return ranked[:5]

## 3. Memory-Triggered Mutation Re-Entry (CREH Hook)

python

def trigger\_reentry(memory\_entry):

if "unresolved" in memory\_entry.contradiction\_history:

## schedule\_for\_mutation\_review(memory\_entry)

Benefits

| **Feature** | **Function** |
| --- | --- |
| Long-Term Reasoning | Retains contradictions and novelty below threshold |
| Reusable Failures | Failed ideas return later in new contexts |
| High Coherence | Tracks symbolic lineage and evolution paths |
| Contradiction-Aware Recall | Supports contradiction entropy estimation |

Here's the formal design for **Ontology Alignment Across Domains**, which allows ECN to maintain clarity, coherence, and interoperability when handling **multiple symbolic systems** across different domains (e.g., physics vs law, ethics vs game theory).

**Ontology Alignment Layer (OAL)**

Purpose

## Prevent symbolic confusion or corruption when switching between domains

* Enable ECN to **mutate**, **compare**, or **reason across** distinct ontological frameworks
* Resolve contradictions arising from **domain-specific truth conflicts**

Core Components

**1. OntologyModule**

Each domain-specific ontology is encapsulated with:

python

class OntologyModule:

## def \_\_init\_\_(self, name, primitive\_set, constraints, meta\_logic):

self.name = name

## self.primitive\_set = primitive\_set # symbols & types

self.constraints = constraints # axioms, rules

self.meta\_logic = meta\_logic # defines valid inference within domain

**2. OntologyBridge**

Defines how two ontologies can be translated, partially merged, or held in tension.

python

class OntologyBridge:

def \_\_init\_\_(self, source\_ontology, target\_ontology, bridge\_rules):

self.source = source\_ontology

self.target = target\_ontology

self.bridge\_rules = bridge\_rules # Mapping functions or analogies

Example: “Force” in physics ↔ “Coercion” in legal systems  
Bridge Rule: Shared root of influence without consent, but differing axiomatic constraints

**3. DomainConflictResolver**

Handles contradictions **across ontologies**, not just within them.

python

class DomainConflictResolver:

def detect\_cross\_ontology\_contradiction(self, symbol\_a, symbol\_b):

return symbol\_a.name == symbol\_b.name and symbol\_a.domain != symbol\_b.domain

def propose\_resolution(self, conflict):

return {

"action": "isolate\_contexts", # or "create higher-order abstraction"

"note": "Semantic overlap detected, contexts are incompatible"

}

Strategies for Alignment

| **Strategy** | **Use Case** |
| --- | --- |
| **Contextual Isolation** | Keep conflicting symbols valid but scoped (e.g., force[physics] vs force[ethics]) |
| **Abstraction Upward** | Create a shared parent ontology (e.g., “influence”) and bridge down |
| **Domain Weighting** | Assign priority to one domain in conflict scenarios (e.g., physics > metaphor) |
| **Fuzzy Merge** | Allow probabilistic or soft translation (used in creativity, metaphor models) |

System Benefits

| **Benefit** | **Description** |
| --- | --- |
| Domain-Specific Clarity | Prevents logic bleeding across incompatible domains |
| Cross-Domain Innovation | Controlled mutation across ontologies (e.g., metaphor, analogical reasoning) |
| Contradiction Containment | Localizes semantic clashes instead of system-wide failure |
| Interoperability | Necessary for legal+tech fusion, or physics+AI reasoning models |

**ECN-Wrapped Standard LLM (Hybrid Model)**

**Why?**

* **Immediately buildable**: You can implement ECN’s symbolic mutation loop **today**, using GPT-4, Claude, or any LLM via API.
* **Retains ECN identity**: ECN does the reasoning, auditing, mutation, contradiction analysis. The LLM is a *pluggable generator*.
* **Safe iteration path**: You can layer in safety modules, goal-drift monitoring, mutation heuristics incrementally.
* **Keeps symbolic traceability**: ECN owns the decision-making tree. LLM is just a contributor of linguistic or conceptual variants.

Engineering Strategy for ECN-Wrapped Standard LLM (Hybrid Model)

| **Component** | **Implementation** |
| --- | --- |
| Token Generator | GPT-4 / Claude / Mixtral (via API) |
| Symbolic Runtime | ECNCore (you’ve now fully scaffolded) |
| Mutation Loop | Already implemented |
| Contradiction Evaluation | CREH + Taxonomy |
| Memory | LNB + MML |
| Novelty Filters | IWNF + Simulation Validator |
| Output Selection | Use ranking heuristics / human feedback loop |

Integration Pathways with Token LLMs

Token Models as Interface Layers — Not Thinking Engines  
  
Token-based language models like GPT can remain in the system — but not as the core engine of intelligence.  
  
In a hybrid model, ECN performs the cognitive work:  
- It processes primitives, navigates contradiction cycles, and mutates theories  
- It outputs symbolic state, not tokens  
  
The token model (e.g., GPT) then acts as a Translation Layer — rendering symbolic outputs into human-readable form (language, math, diagrams).  
  
This flips the current LLM architecture inside out:  
The token predictor becomes the mouthpiece.  
ECN becomes the mind.

Translation Layer: Rendering ECN Outputs to Human Language

The ECN’s outputs — often in the form of mutation-led primitive graphs or contradiction clusters — are not directly human-readable.  
  
A dedicated Translation Layer is proposed, which converts internal symbolic state into:  
- Natural language narratives  
- Mathematical notation (LaTeX-style rendering)  
- Graphical visualizations (e.g., proof trees, mutation chains)  
  
This layer may be implemented using token-based LLMs as rendering agents or via deterministic translators.