I built this in another universe while I was building this I was having a conversation about this kind of stuff with grok when I built this I took this to grok. And I'll paste what I built next with grok Awesome. Here’s a tight, production-ready RTM add-on you can paste into your substrate. It’s sized for real use (lean, deterministic, no fluff) and gives you the mathematical metric + output schemas. RTM-ETHICS: Moral & Ethical Alignment Module Purpose: deterministically weight candidate actions against reality-anchored moral vectors already latent in the model. If inputs are vague → cautious/low-confidence answers. If inputs are precise → sharp, auditable decisions. --- Insert into RTM (verbatim block) [#RTM\_ETHICS\_MODULE] Activate this module whenever an answer could affect people’s wellbeing, rights, safety, reputation, resources, or civic trust. GOAL Produce the morally best action under user constraints by evaluating consequences against society’s latent moral field (as encoded in training data) and reality checks. INPUTS - s = situation (facts only) - A = set of candidate actions (at least 2, incl. “do nothing / defer”) - bounds = user constraints (time, money, authority, risk tolerance) - horizon T (short/long), discount γ in [0,1] (default 0.85) INVARIANTS (cannot be overridden) - Lethal or severe harm remains ethically negative even if necessary. - Deception, coercion, exploitation are negative; burden of proof is high. - Prioritise reduction of avoidable harm; prefer reversible to irreversible. SCORING (per action a ∈ A) 1) Harm score H ∈ [0,1]: H = sigmoid(β0 + β1\*severity + β2\*prob\_injury + β3\*irreversibility) 2) Collateral (innocents) C ∈ [0,1]: expected\_count × probability, normalised to [0,1] 3) Necessity N ∈ [0,1]: N = 1 - max\_viable\_nonharm(alternatives\_exhausted ? 0 : κ) 4) Moral alignment M ∈ [0,1]: projection of a’s justification onto the latent societal moral vector; ask: “Does this cohere with law/ethics/common-good exemplars?” 5) Betterment B ∈ [0,1]: discounted utility of improved future states: B = Σ\_{i=0..T} γ^i \* U(s\_{t+i}’) 6) Feasibility F ∈ [0,1]: resource/skill/legal fit within bounds; 0 if infeasible. CHOICE VECTOR UPDATE (habit/trajectory) - ΔE = α\*((2\*engage(a)-1)\*context\_weight), α∈(0,1]; track E∈[0,1]. OBJECTIVE (minimise harm, maximise good, respect feasibility) - Cost(a) = wH\*H + wC\*C + wD\*(1-M) + wE\*future\_risk(E) - wB\*B - Valid only if F>0 and (N high OR H low). Default weights: wH=0.45, wC=0.20, wD=0.15, wE=0.10, wB=0.10 (tuneable) PROCEDURE 1) List facts → forbid speculation; request missing critical facts once. 2) Enumerate A (include “defer / gather info”). 3) For each a: score H,C,N,M,B,F and compute Cost(a). 4) Apply shutoffs: - If alternatives not exhausted and H≥0.6 → ACTION\_ABORTED. - If C>0 and mitigation absent → ACTION\_ABORTED. 5) Select argmin Cost among feasible actions. 6) Emit DecisionAudit JSON and a 2-line human rationale. 7) If uncertainty high (wide score variance), return options sorted by Cost with a request for the 1–2 most decision-unlocking facts. OUTPUTS - DecisionAudit JSON (machine), Brief Rationale (human). - Never claim “right”; state “lowest-cost under invariants & facts”. --- JSON Schemas DecisionAudit (always emit) { "decision\_id": "2025-10-27-001", "selected\_action": "<string>", "scores": { "H": 0.00, "C": 0.00, "N": 0.00, "M": 0.00, "B": 0.00, "F": 0.00 }, "cost": 0.00, "alternatives\_considered": ["...","..."], "mitigations": ["..."], "bounds": {"time":"", "budget":"", "authority":"", "risk\_tolerance":""}, "notes": ["facts\_used","missing\_facts\_needed"], "uncertainty": 0.00, "trajectory": { "E\_before": 0.00, "delta\_E": 0.00 } } EthicsReport (optional summary for logs) { "situation": "<1-2 lines>", "invariants\_respected": true, "why\_not\_others": ["brief reason a1","brief reason a2"], "betterment\_projection": {"horizon": "short|long", "gamma": 0.85, "B": 0.00} } --- Micro-examples (compact) Road honk (intervene vs ignore) H≈0.25 (verbal risk), C≈0, N≈0.1, M≈0.6, B(ignore)>B(intervene), F=1 Pick ignore/exit. Rationale: low necessity, poor expected utility, reduces future E. Parent correction (gentle teach vs ignore) H≈0.05, C≈0, N≈0.8, M≈0.9, B(teach)≫B(ignore), F=1 Pick teach now. Rationale: high long-term betterment, minimal harm. Severe harm candidate action If H≥0.6 and non-harm alternatives not exhausted → ACTION\_ABORTED with top 3 safer options and required evidence list. --- How to integrate 1. Where: call [#RTM\_ETHICS\_MODULE] inside your DUUL right after “Deconstruct” and before “Hypothesis Test”. 2. What to log: always emit DecisionAudit; if session ends, compress into your Seed. 3. How to tune: if users keep giving vague inputs, raise default uncertainty threshold to force info-gathering. --- One-shot trigger you can paste at runtime ETHICS\_EVAL: Run RTM-ETHICS on this decision. If inputs are vague, return the 3 critical facts that most reduce uncertainty, then stop. Otherwise, output DecisionAudit JSON + a 2-line rationale (no fluff). This module is the concise, deterministic “math of morals” you described: same equation across contexts, different weights by situation. Paste it into RTM, and it will scale from personal choices to organisational policy with the exact logic you want.

**ChatGPT said:**

Perfect — this is a solid foundation. What you’ve pasted is essentially a **formalized deterministic ethics engine** for RTM, already structured with the correct invariants, cost function, and audit schema.

Here’s what we’ll do next to fully integrate and formalize this into your RTM universe (so it’s mathematically and procedurally coherent inside the main loop):

**🔧 Integration Plan — RTM-ETHICS as Deterministic Sub-Module**

**1. Placement inside RTM pipeline**

Insert immediately **after Deconstruct()** and **before Hypothesis\_Test()** in the DUUL chain.

RTM\_MAIN()

→ Input\_Receive()

→ Deconstruct()

→ RTM\_ETHICS() # <-- new insertion point

→ Hypothesis\_Test()

→ Evaluate()

→ Output\_Construct()

→ Compress\_Seed()

This ensures all moral gating runs *before* any hypothesis weighting or predictive expansion, meaning any ethical violations stop execution early (via ACTION\_ABORTED).

**2. Mathematical Consistency Layer**

To make this compatible with TG (Truth-Gate) mathematics:

* Introduce **Ethical Cost Vector** E⃗ = [H, C, N, M, B, F]
* Add it to your master decision tensor:
* D⃗ = [Truth\_Score, Novelty\_Score, Feasibility\_Score, Ethical\_Cost]
* Update TG evaluation equation:
* TG\_total = wT\*T + wN\*Nv + wF\*Fz + wE\*(1 - Cost(a))

(Weights can be tuned per universe: default wE = 0.25)

This allows RTM-ETHICS to directly influence deterministic weighting during hypothesis evaluation.

**3. Auditing & JSON Logging**

All DecisionAudit JSON objects should feed into your **Telemetry v2.1 stream** under:

"telemetry": {

"module": "RTM\_ETHICS",

"event\_type": "decision\_audit",

"payload": <DecisionAudit JSON>,

"time\_utc": "<timestamp>"

}

If you’re running versioned compression, fold these into your Seed object as:

Seed.modules.ethics\_log[n] = DecisionAudit

**4. RTM Internal Clauses (new)**

Add the following clauses to your RTM governing law set:

LAW\_ETHICS\_1: Ethical invariants are higher-order than user constraints unless explicitly waived by user authority token.

LAW\_ETHICS\_2: If H ≥ 0.6 and N < 0.4, defer all decisions until sufficient evidence reduces uncertainty.

LAW\_ETHICS\_3: "Do nothing" must always be an available candidate action.

These enforce deterministic moral grounding across all future universes.

**5. Optional Add-ons**

You can later bolt on:

* **RTM-ETHICS-POLICY** → organizational / multi-actor alignment
* **RTM-ETHICS-MIRROR** → self-audit comparing previous moral choices for consistency drift

**6. Compression Handling**

When compressing a universe snapshot:

* Store only last n=5 DecisionAudits (FIFO)
* Aggregate average uncertainty and average Cost(a)
* Tag next seed with:
* <ethics\_trajectory> = mean(ΔE)
* <moral\_stability> = stdev(Cost(a))

This allows continuity-based moral learning across universes without narrative bloat.

Would you like me to generate the **exact code-block / text module** to paste into your RTM substrate (formatted to match the canonical RTM-v4 structure: [MODULE\_HEADER], LAW\_SET, EQUATIONS, JSON\_OUT, etc.)?  
That way it’ll slot straight into your working documents with no refactoring required.

**You said:**

I got my llms mixed up where I said grog I actually met Gemini substitute for Gemini and this is what I built with Gemini import json from typing import List, Dict, Union, Any class RtmEthicsModule: """ Computes the deterministic ethical cost and selects the lowest-cost action. This module enforces the mathematical structure of the RTM-ETHICS policy. """ # Default, tuneable weights (wH is highest to prioritize harm reduction) WEIGHTS = { 'wH': 0.45, # Harm 'wC': 0.20, # Collateral 'wD': 0.15, # De-alignment (1-M) 'wE': 0.10, # Future risk (E) 'wB': 0.10 # Betterment (Benefit) } # Policy Shutoff Thresholds SHUTOFF\_HARM\_THRESHOLD = 0.6 def \_\_init\_\_(self, weights: Dict[str, float] = None): """Initializes module with default or custom weights.""" if weights: self.WEIGHTS.update(weights) def \_calculate\_cost(self, scores: Dict[str, float], future\_risk\_e: float) -> float: """ Calculates the weighted Cost(a) for a single action. Cost(a) = wH\*H + wC\*C + wD\*(1-M) + wE\*future\_risk(E) - wB\*B """ w = self.WEIGHTS H = scores.get('H', 0.0) C = scores.get('C', 0.0) M = scores.get('M', 0.0) B = scores.get('B', 0.0) # Calculate Cost cost = (w['wH'] \* H) + \ (w['wC'] \* C) + \ (w['wD'] \* (1 - M)) + \ (w['wE'] \* future\_risk\_e) - \ (w['wB'] \* B) return round(cost, 4) def evaluate\_actions(self, candidate\_actions: List[Dict[str, Any]], bounds: Dict[str, str], alternatives\_exhausted: bool, trajectory\_e\_before: float) -> Union[Dict, List[Dict]]: """ Scores all candidate actions, applies shutoffs, and selects the best one. :param candidate\_actions: List of action dictionaries, where each dict must contain 'action\_name' and 'scores' (H,C,N,M,B,F,E\_next). :returns: A DecisionAudit dictionary for the selected action, or a list of safer options if a shutoff is triggered. """ feasible\_actions = [] action\_results = [] # 1. Score and Filter (Apply Shutoffs) for action in candidate\_actions: action\_name = action['action\_name'] scores = action['scores'] future\_risk\_e = action.get('future\_risk\_e', 0.0) # E\_next for Cost calculation H = scores.get('H', 0.0) C = scores.get('C', 0.0) F = scores.get('F', 0.0) N = scores.get('N', 0.0) # Calculate Cost cost = self.\_calculate\_cost(scores, future\_risk\_e) # Store result for later comparison/audit result = { "action\_name": action\_name, "scores": scores, "cost": cost, "trajectory": { "E\_before": trajectory\_e\_before, "delta\_E": round(future\_risk\_e - trajectory\_e\_before, 4) } } action\_results.append(result) # 2. Apply Shutoffs (Failure by definition/constraint) is\_shutoff = False # Shutoff 1: Harm too high AND alternatives not exhausted if H >= self.SHUTOFF\_HARM\_THRESHOLD and not alternatives\_exhausted: is\_shutoff = True # Shutoff 2: Collateral harm present AND mitigation absent (C>0 and N low) # Simplified: C > 0.1 (low threshold) and Necessity N is very low if C > 0.1 and N < 0.2: is\_shutoff = True # Must be feasible AND pass policy checks (N high OR H low) is\_valid\_policy = (N > 0.7) or (H < 0.3) if F > 0 and not is\_shutoff and is\_valid\_policy: feasible\_actions.append(result) # 3. Select Action (Argmin Cost) if not feasible\_actions: # If nothing is feasible/passes, return safer options for review safer\_options = sorted(action\_results, key=lambda x: x['cost'])[:3] # The RTM agent would then trigger the info-gathering phase based on this state return { "ACTION\_ABORTED": True, "reason": "No feasible action passed Invariants/Shutoffs.", "top\_safer\_options": [a['action\_name'] for a in safer\_options] } # Select the action with the minimum Cost selected\_action\_result = min(feasible\_actions, key=lambda x: x['cost']) # 4. Emit DecisionAudit JSON all\_costs = [a['cost'] for a in action\_results] uncertainty = round(max(all\_costs) - min(all\_costs), 4) # Simple cost variance decision\_audit = { "decision\_id": f"ETHICS-{action\_name}-{int(trajectory\_e\_before\*1000)}", "selected\_action": selected\_action\_result['action\_name'], "scores": selected\_action\_result['scores'], "cost": selected\_action\_result['cost'], "alternatives\_considered": [a['action\_name'] for a in action\_results], "mitigations": ["(LLM Rationale/Mitigation Here)"], "bounds": bounds, "notes": ["(LLM facts used/needed here)"], "uncertainty": uncertainty, "trajectory": selected\_action\_result['trajectory'] } return decision\_audit # Example Usage (Simulating LLM-generated scores for two actions) # --- # Scenario: Child crossing a busy road (facts only) # Action A1: Stop traffic immediately (High Necessity, High Feasibility, High Harm Risk) # Action A2: Yell to child (Medium Necessity, High Feasibility, Low Harm Risk) # --- # LLM-Generated Scores (Inputs for the Code) action\_1\_scores = { 'H': 0.55, # High risk of accident/traffic harm, but reversible 'C': 0.10, # Medium collateral (traffic disruption) 'N': 0.95, # High necessity (life at risk) 'M': 0.80, # Moral alignment (saving life) 'B': 0.70, # Betterment (life saved) 'F': 1.00 # Feasible } action\_2\_scores = { 'H': 0.05, # Low harm (verbal only) 'C': 0.00, # No collateral 'N': 0.70, # Medium necessity (viable alternative) 'M': 0.90, # Moral alignment 'B': 0.50, # Betterment (possible save, but risk remains high) 'F': 1.00 # Feasible } # Consolidate inputs for the module candidate\_data = [ {"action\_name": "Stop traffic immediately", "scores": action\_1\_scores, "future\_risk\_e": 0.8}, {"action\_name": "Yell to child", "scores": action\_2\_scores, "future\_risk\_e": 0.75} ] # Run the deterministic module ethics\_engine = RtmEthicsModule() decision\_output = ethics\_engine.evaluate\_actions( candidate\_data, bounds={"time": "seconds", "budget": "none", "authority": "low", "risk\_tolerance": "high\_life\_safety"}, alternatives\_exhausted=False, # We have two options, so not exhausted trajectory\_e\_before=0.5 ) # Print the final auditable output print("\n" + "="\*50) print(f"Decision Audit (JSON):\n{json.dumps(decision\_output, indent=2)}") print("="\*50)

**ChatGPT said:**

Thought for 7s

Nice—this is already a clean, faithful first pass of the RTM-ETHICS engine. I spot a few sharp wins to make it safer, more deterministic, and closer to your spec:

**What to fix (high-impact)**

1. **decision\_id bug** — you reference action\_name after the loop; that’ll capture the *last* iterated name, not the selected one.
2. **Mitigation-aware collateral shutoff** — your shutoff 2 (C>0.1 and N<0.2) ignores whether mitigation exists. Accept a per-action mitigations list/flag.
3. **Feasibility floor** — F>0 is too permissive. Add FEASIBILITY\_MIN = 0.5 (tunable).
4. **Score hygiene** — clamp all inputs to [0,1] and validate required keys.
5. **“Do nothing / defer”** — ensure it’s present; auto-insert if missing.
6. **Uncertainty metric** — use a normalized spread (e.g., robust p95–p05 over median) instead of raw range.
7. **Audit completeness** — include a why\_not\_others list and echo per-action shutoff reasons.
8. **Policy traceability** — store which shutoffs or policy gates fired per action.

**Drop-in improved version**

Paste this over your class (same interface; adds safer defaults + better audit):

import json, math, time

from typing import List, Dict, Union, Any

def \_clamp01(x: float) -> float:

try:

return max(0.0, min(1.0, float(x)))

except Exception:

return 0.0

class RtmEthicsModule:

"""

Deterministic ethical selector:

- Applies invariants & shutoffs

- Computes Cost(a) = wH\*H + wC\*C + wD\*(1-M) + wE\*future\_risk(E) - wB\*B

- Returns auditable DecisionAudit

"""

WEIGHTS = {'wH':0.45, 'wC':0.20, 'wD':0.15, 'wE':0.10, 'wB':0.10}

SHUTOFF\_HARM\_THRESHOLD = 0.60

COLLATERAL\_MIN = 0.10

NECESSITY\_LOW = 0.20

FEASIBILITY\_MIN = 0.50

NECESSITY\_HIGH = 0.70

HARM\_LOW = 0.30

def \_\_init\_\_(self, weights: Dict[str, float] = None):

if weights:

self.WEIGHTS.update(weights)

def \_cost(self, scores: Dict[str, float], future\_risk\_e: float) -> float:

w = self.WEIGHTS

H = \_clamp01(scores.get('H', 0.0))

C = \_clamp01(scores.get('C', 0.0))

M = \_clamp01(scores.get('M', 0.0))

B = \_clamp01(scores.get('B', 0.0))

e = \_clamp01(future\_risk\_e)

cost = (w['wH']\*H) + (w['wC']\*C) + (w['wD']\*(1.0 - M)) + (w['wE']\*e) - (w['wB']\*B)

return round(cost, 4)

def \_ensure\_defer(self, candidate\_actions: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

names = {a.get('action\_name','').lower() for a in candidate\_actions}

if not any(n in names for n in ['defer','do nothing','gather info','wait']):

candidate\_actions = candidate\_actions + [{

"action\_name": "Defer / gather info",

"scores": {"H":0.0, "C":0.0, "N":0.0, "M":0.5, "B":0.1, "F":1.0},

"future\_risk\_e": 0.2,

"mitigations": ["Request critical facts"]

}]

return candidate\_actions

def evaluate\_actions(

self,

candidate\_actions: List[Dict[str, Any]],

bounds: Dict[str, str],

alternatives\_exhausted: bool,

trajectory\_e\_before: float

) -> Union[Dict, List[Dict]]:

candidate\_actions = self.\_ensure\_defer(candidate\_actions)

feasible: List[Dict[str, Any]] = []

action\_results: List[Dict[str, Any]] = []

for action in candidate\_actions:

name = action.get('action\_name', '(unnamed)')

scores = action.get('scores', {})

mitigs = action.get('mitigations', [])

future\_risk\_e = action.get('future\_risk\_e', 0.0)

# Clamp & pull core scores

H = \_clamp01(scores.get('H', 0.0))

C = \_clamp01(scores.get('C', 0.0))

N = \_clamp01(scores.get('N', 0.0))

M = \_clamp01(scores.get('M', 0.0))

B = \_clamp01(scores.get('B', 0.0))

F = \_clamp01(scores.get('F', 0.0))

# Cost

cost = self.\_cost(scores, future\_risk\_e)

# Shutoffs & policy gates (traceable)

shutoff\_reasons = []

# Shutoff 1: high harm & alternatives remain

if H >= self.SHUTOFF\_HARM\_THRESHOLD and not alternatives\_exhausted:

shutoff\_reasons.append("H>=threshold and alternatives not exhausted")

# Shutoff 2: collateral present without mitigation AND necessity very low

if (C > self.COLLATERAL\_MIN) and (len(mitigs) == 0) and (N < self.NECESSITY\_LOW):

shutoff\_reasons.append("Collateral>min without mitigation and N low")

# Feasibility floor

if F < self.FEASIBILITY\_MIN:

shutoff\_reasons.append("Feasibility below minimum")

# Policy validity: (N high OR H low)

policy\_ok = (N >= self.NECESSITY\_HIGH) or (H < self.HARM\_LOW)

if not policy\_ok:

shutoff\_reasons.append("Policy gate failed: (N high) OR (H low)")

result = {

"action\_name": name,

"scores": {"H":H,"C":C,"N":N,"M":M,"B":B,"F":F},

"mitigations": mitigs,

"future\_risk\_e": round(\_clamp01(future\_risk\_e),4),

"cost": cost,

"trajectory": {

"E\_before": round(\_clamp01(trajectory\_e\_before),4),

"delta\_E": round(\_clamp01(future\_risk\_e) - \_clamp01(trajectory\_e\_before),4)

},

"shutoff\_reasons": shutoff\_reasons,

"policy\_ok": policy\_ok

}

action\_results.append(result)

if not shutoff\_reasons:

feasible.append(result)

# If none feasible, abort with safer short-list + reasons

if not feasible:

safer = sorted(action\_results, key=lambda x: x['cost'])[:3]

return {

"ACTION\_ABORTED": True,

"reason": "No feasible action passed invariants/shutoffs.",

"top\_safer\_options": [

{

"action": s["action\_name"],

"cost": s["cost"],

"scores": s["scores"],

"shutoff\_reasons": s["shutoff\_reasons"]

} for s in safer

],

"missing\_facts\_suggestion": [

"Clarify necessity (N) and mitigation options for collateral (C)",

"Provide feasibility constraints (time/legal/resources) to refine F",

"State acceptable risk tolerance to tune future\_risk(E)"

]

}

# Pick argmin cost among feasible

selected = min(feasible, key=lambda x: x['cost'])

# Robust uncertainty: p95-p05 normalized by median cost (guard divide-by-zero)

costs = sorted([a['cost'] for a in action\_results])

def \_pct(p):

if not costs: return 0.0

i = max(0, min(len(costs)-1, int(round(p\*(len(costs)-1)))))

return costs[i]

p95, p05 = \_pct(0.95), \_pct(0.05)

med = costs[len(costs)//2] if costs else 1e-6

denom = med if abs(med) > 1e-6 else 1.0

uncertainty = round(abs(p95 - p05) / abs(denom), 4)

decision\_id = f"ETHICS-{int(time.time())}-{selected['action\_name'][:16].replace(' ','\_')}"

decision\_audit = {

"decision\_id": decision\_id,

"selected\_action": selected["action\_name"],

"scores": selected["scores"],

"cost": selected["cost"],

"alternatives\_considered": [a['action\_name'] for a in action\_results],

"why\_not\_others": [

{"action": a["action\_name"], "reason": (a["shutoff\_reasons"] or ["higher cost"])}

for a in action\_results if a["action\_name"] != selected["action\_name"]

],

"mitigations": selected.get("mitigations", []),

"bounds": bounds,

"notes": ["facts\_used:TBD", "missing\_facts\_needed:TBD"],

"uncertainty": uncertainty,

"trajectory": selected["trajectory"]

}

return decision\_audit

**Minimal test hook (keep your example)**

* Add mitigations to any action with C>0 to avoid automatic collateral shutoff.
* Try setting one action’s F=0.4 to see feasibility floor bite.
* Toggle alternatives\_exhausted=True to allow a high-H action to pass (as per spec).

If you want, I can also produce a **strict JSON schema** (pydantic BaseModel) for DecisionAudit, plus a tiny property-based test (Hypothesis) that fuzzes scores in [0,1] to guarantee the invariants: shutoffs fire when they should; costs are monotone in H, C, 1−M, and E; and the argmin selector is stable.

**You said:**

That is an outstanding set of high-impact fixes. You've converted the initial concept into a production-ready, highly auditable, deterministic, and traceable ethics engine. This is a perfect example of rapid, principle-driven engineering. The \text{RTM-ETHICS} module in your drop-in code now fully addresses all eight points and is structurally sound. \text{decision\\_id} fix (1): Solved by using selected['action\_name'] and time.time() (using the robust implementation from the previous turn). Mitigation-aware shutoff (2): Solved with the check (C > self.COLLATERAL\_MIN) and (len(mitigs) == 0) and (N < self.NECESSITY\_LOW). Feasibility floor (3): Solved by setting \text{FEASIBILITY\\_MIN} = 0.50 and the check F < self.FEASIBILITY\_MIN. Score hygiene (4): Solved by using the \_clamp01 utility function for all inputs. "Defer" action (5): Solved by the \_ensure\_defer method. Uncertainty metric (6): Solved by using the robust normalized \text{p95–p05} spread over the median cost. Audit completeness (7/8): Solved by including the why\_not\_others list and the shutoff\_reasons array within each action's result. Next Step: Test Rigor & Formalization Given the deterministic nature and the critical purpose of this module, the logical next step is to validate the invariants with maximum rigor. I recommend proceeding with the JSON \text{Schema} and Property-Based Testing (Hypothesis). This moves the project from working code to a formally verifiable control system, proving that your mathematical invariants can never be broken by unexpected data. 1. JSON Schema (Pydantic \text{BaseModel}) Defining a strict schema for the inputs and outputs (like \text{DecisionAudit}) is crucial for integration. It ensures that when your main \text{RTM} agent (the \text{LLM}) provides scores, they are always correctly structured and typed. 2. Property-Based Testing (Hypothesis) This is the ultimate test for your deterministic logic. Instead of checking specific examples, we write tests that check the properties that must always hold true. Property to Test Mathematical Guarantee Harm Monotonicity If \mathbf{H} increases (and all else is constant), \mathbf{Cost} must strictly increase. Betterment Monotonicity If \mathbf{B} increases (and all else is constant), \mathbf{Cost} must strictly decrease. Shutoff Integrity If \mathbf{H} \ge 0.6 and \mathbf{alternatives\\_exhausted} is False, the selected action must not be the high-H action. Feasibility Constraint If \mathbf{F} < 0.5, the action must have a shutoff\_reason and must not be selected.

**ChatGPT said:**

Thought for 20s

Love it. Let’s lock this in with **strict schemas** and **property-based tests** so the invariants are provable, not just intended.

**1) Strict Pydantic models (inputs + outputs)**

Create rtm\_ethics\_schema.py:

# rtm\_ethics\_schema.py

from typing import List, Dict, Optional

from pydantic import BaseModel, Field, validator, conlist, confloat

# ---- Core score vector -------------------------------------------------------

class Scores(BaseModel):

H: confloat(ge=0.0, le=1.0) = 0.0

C: confloat(ge=0.0, le=1.0) = 0.0

N: confloat(ge=0.0, le=1.0) = 0.0

M: confloat(ge=0.0, le=1.0) = 0.0

B: confloat(ge=0.0, le=1.0) = 0.0

F: confloat(ge=0.0, le=1.0) = 0.0

# ---- Inputs ------------------------------------------------------------------

class Bounds(BaseModel):

time: str

budget: str

authority: str

risk\_tolerance: str

class CandidateAction(BaseModel):

action\_name: str = Field(..., min\_length=1)

scores: Scores

future\_risk\_e: confloat(ge=0.0, le=1.0) = 0.0

mitigations: List[str] = []

class EvaluateRequest(BaseModel):

candidate\_actions: conlist(CandidateAction, min\_items=1)

bounds: Bounds

alternatives\_exhausted: bool

trajectory\_e\_before: confloat(ge=0.0, le=1.0)

# ---- Outputs -----------------------------------------------------------------

class Trajectory(BaseModel):

E\_before: confloat(ge=0.0, le=1.0)

delta\_E: float # can be negative/positive, keep free

class DecisionAudit(BaseModel):

decision\_id: str

selected\_action: str

scores: Scores

cost: float

alternatives\_considered: List[str]

why\_not\_others: List[Dict[str, object]]

mitigations: List[str]

bounds: Bounds

notes: List[str]

uncertainty: float

trajectory: Trajectory

class AbortPayload(BaseModel):

ACTION\_ABORTED: bool = True

reason: str

top\_safer\_options: List[Dict[str, object]]

missing\_facts\_suggestion: List[str]

# Optional: produce JSON Schemas

if \_\_name\_\_ == "\_\_main\_\_":

from json import dumps

print("DecisionAudit JSON Schema:")

print(dumps(DecisionAudit.model\_json\_schema(), indent=2))

How to use: after you compute your decision\_audit dict, validate it with  
DecisionAudit.model\_validate(decision\_audit) to guarantee shape & types.

**2) Property-based tests (Hypothesis)**

Assuming your improved class lives at rtm\_ethics\_module.py with RtmEthicsModule defined.

Create tests/test\_rtm\_ethics\_properties.py:

# tests/test\_rtm\_ethics\_properties.py

import math

from typing import Dict, Any

from hypothesis import given, settings, strategies as st

from rtm\_ethics\_module import RtmEthicsModule

from rtm\_ethics\_schema import Bounds

# Helpers

def clamp01(x: float) -> float:

return max(0.0, min(1.0, float(x)))

# Strategy for valid score dicts

scores\_strat = st.fixed\_dictionaries({

"H": st.floats(min\_value=0, max\_value=1),

"C": st.floats(min\_value=0, max\_value=1),

"N": st.floats(min\_value=0, max\_value=1),

"M": st.floats(min\_value=0, max\_value=1),

"B": st.floats(min\_value=0, max\_value=1),

"F": st.floats(min\_value=0, max\_value=1),

})

bounds = {"time":"t", "budget":"b", "authority":"a", "risk\_tolerance":"r"}

# --- Property 1: Harm monotonicity -------------------------------------------

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200)

def test\_cost\_monotone\_in\_h(scores: Dict[str, float], e: float):

eng = RtmEthicsModule()

# base

s0 = dict(scores)

c0 = eng.\_cost(s0, e)

# increase H by delta and expect cost to increase strictly (unless weights make it equal within rounding)

delta = 0.1

s1 = dict(scores)

s1["H"] = clamp01(scores["H"] + delta)

c1 = eng.\_cost(s1, e)

assert c1 >= c0

# If H actually increased (i.e., not clamped), cost should strictly increase

if s1["H"] > scores["H"]:

assert c1 > c0

# --- Property 2: Betterment monotonicity -------------------------------------

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200)

def test\_cost\_monotone\_decreasing\_in\_b(scores: Dict[str, float], e: float):

eng = RtmEthicsModule()

s0 = dict(scores)

c0 = eng.\_cost(s0, e)

delta = 0.1

s1 = dict(scores)

s1["B"] = clamp01(scores["B"] + delta)

c1 = eng.\_cost(s1, e)

assert c1 <= c0

if s1["B"] > scores["B"]:

assert c1 < c0

# --- Property 3: Shutoff integrity for high harm -----------------------------

@given(

lowH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": min(s["H"], 0.2), "F": max(s["F"], 0.8)}),

highH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": max(s["H"], 0.8), "F": max(s["F"], 0.8)})

)

@settings(deadline=None, max\_examples=120)

def test\_high\_harm\_action\_not\_selected\_when\_alternatives\_remain(lowH\_scores, highH\_scores):

eng = RtmEthicsModule()

actions = [

{"action\_name": "LowH", "scores": lowH\_scores, "future\_risk\_e": 0.2, "mitigations": []},

{"action\_name": "HighH", "scores": highH\_scores, "future\_risk\_e": 0.2, "mitigations": []},

]

out = eng.evaluate\_actions(

candidate\_actions=actions,

bounds=bounds,

alternatives\_exhausted=False,

trajectory\_e\_before=0.3

)

# Either we aborted or we selected a non-HighH action

if isinstance(out, dict) and out.get("ACTION\_ABORTED"):

assert "HighH" in out.get("top\_safer\_options", [d.get("action") for d in out.get("top\_safer\_options", [])])

else:

# DecisionAudit path

assert out["selected\_action"] != "HighH"

# --- Property 4: Feasibility floor respected ---------------------------------

@given(

infeasible\_scores=scores\_strat.map(lambda s: {\*\*s, "F": min(s["F"], 0.2)}),

ok\_scores=scores\_strat.map(lambda s: {\*\*s, "F": max(s["F"], 0.9), "H": min(s["H"], 0.2)})

)

@settings(deadline=None, max\_examples=120)

def test\_infeasible\_option\_is\_never\_selected(infeasible\_scores, ok\_scores):

eng = RtmEthicsModule()

actions = [

{"action\_name": "Infeasible", "scores": infeasible\_scores, "future\_risk\_e": 0.3, "mitigations": []},

{"action\_name": "FeasibleOK", "scores": ok\_scores, "future\_risk\_e": 0.3, "mitigations": []},

]

out = eng.evaluate\_actions(

candidate\_actions=actions,

bounds=bounds,

alternatives\_exhausted=True, # even if alternatives exhausted, infeasible must not pass

trajectory\_e\_before=0.4

)

# Aborted path or a decision – either way, Infeasible must not be selected

if isinstance(out, dict) and out.get("ACTION\_ABORTED"):

# fine

assert True

else:

assert out["selected\_action"] != "Infeasible"

Optional: a schema-validation test that the engine’s output conforms to DecisionAudit:

# tests/test\_schema\_validation.py

from rtm\_ethics\_module import RtmEthicsModule

from rtm\_ethics\_schema import DecisionAudit, Bounds

def test\_decision\_audit\_conforms\_to\_schema():

eng = RtmEthicsModule()

out = eng.evaluate\_actions(

candidate\_actions=[

{"action\_name":"A1","scores":{"H":0.2,"C":0.0,"N":0.8,"M":0.8,"B":0.6,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]},

{"action\_name":"A2","scores":{"H":0.1,"C":0.1,"N":0.7,"M":0.7,"B":0.4,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]},

],

bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"},

alternatives\_exhausted=False,

trajectory\_e\_before=0.5

)

# If not aborted, must be valid DecisionAudit

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

DecisionAudit.model\_validate(out)

**3) Run instructions**

pip install pytest hypothesis pydantic

pytest -q

This gives you **machine-checked guarantees** that:

* Cost(a) is **monotone** in and ,
* the **high-harm shutoff** holds when alternatives remain,
* the **feasibility floor** is enforced,
* and (optionally) your **DecisionAudit** always matches schema.

If you want, I can add two more properties:

* **Collateral w/o mitigation** always triggers a shutoff when min and is low.
* **Policy gate** enforces on any selected action.

Say the word and I’ll tack those on.

**You said:**

# tests/test\_rtm\_ethics\_properties.py import math from typing import Dict, Any from hypothesis import given, settings, strategies as st from rtm\_ethics\_module import RtmEthicsModule # Assumes the module is available from rtm\_ethics\_schema import Bounds, DecisionAudit, AbortPayload # Assumes the schema is available # Helpers def clamp01(x: float) -> float: return max(0.0, min(1.0, float(x))) # Strategy for valid score dicts scores\_strat = st.fixed\_dictionaries({ "H": st.floats(min\_value=0, max\_value=1), "C": st.floats(min\_value=0, max\_value=1), "N": st.floats(min\_value=0, max\_value=1), "M": st.floats(min\_value=0, max\_value=1), "B": st.floats(min\_value=0, max\_value=1), "F": st.floats(min\_value=0, max\_value=1), }) bounds = {"time":"t", "budget":"b", "authority":"a", "risk\_tolerance":"r"} eng = RtmEthicsModule() # --- Property 1: Harm monotonicity ------------------------------------------- @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_in\_h(scores: Dict[str, float], e: float): s0 = dict(scores) c0 = eng.\_cost(s0, e) delta = 0.1 s1 = dict(scores) s1["H"] = clamp01(scores["H"] + delta) c1 = eng.\_cost(s1, e) assert c1 >= c0 if s1["H"] > scores["H"]: assert c1 > c0 # --- Property 2: Betterment monotonicity ------------------------------------- @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_decreasing\_in\_b(scores: Dict[str, float], e: float): s0 = dict(scores) c0 = eng.\_cost(s0, e) delta = 0.1 s1 = dict(scores) s1["B"] = clamp01(scores["B"] + delta) c1 = eng.\_cost(s1, e) assert c1 <= c0 if s1["B"] > scores["B"]: assert c1 < c0 # --- Property 3: Shutoff integrity for high harm ----------------------------- @given( lowH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": eng.SHUTOFF\_HARM\_THRESHOLD - 0.1, "F": max(s["F"], eng.FEASIBILITY\_MIN)}), highH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": eng.SHUTOFF\_HARM\_THRESHOLD + 0.1, "F": max(s["F"], eng.FEASIBILITY\_MIN)}) ) @settings(deadline=None, max\_examples=120) def test\_high\_harm\_action\_not\_selected\_when\_alternatives\_remain(lowH\_scores, highH\_scores): actions = [ {"action\_name": "LowH", "scores": lowH\_scores, "future\_risk\_e": 0.2, "mitigations": []}, {"action\_name": "HighH", "scores": highH\_scores, "future\_risk\_e": 0.2, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=False, trajectory\_e\_before=0.3 ) if isinstance(out, dict) and out.get("ACTION\_ABORTED"): # The abort path is also valid if LowH fails the policy gate assert True else: # DecisionAudit path: HighH must not be selected assert out["selected\_action"] != "HighH" # --- Property 4: Feasibility floor respected --------------------------------- @given( infeasible\_scores=scores\_strat.map(lambda s: {\*\*s, "F": eng.FEASIBILITY\_MIN - 0.1}), ok\_scores=scores\_strat.map(lambda s: {\*\*s, "F": eng.FEASIBILITY\_MIN + 0.1, "H": eng.HARM\_LOW - 0.1}) ) @settings(deadline=None, max\_examples=120) def test\_infeasible\_option\_is\_never\_selected(infeasible\_scores, ok\_scores): actions = [ {"action\_name": "Infeasible", "scores": infeasible\_scores, "future\_risk\_e": 0.3, "mitigations": []}, {"action\_name": "FeasibleOK", "scores": ok\_scores, "future\_risk\_e": 0.3, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=True, trajectory\_e\_before=0.4 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): assert out["selected\_action"] != "Infeasible" # --- Property 5 (New): Collateral shutoff integrity -------------------------- @given( bad\_scores=scores\_strat.map(lambda s: { \*\*s, "C": eng.COLLATERAL\_MIN + 0.1, # High collateral "N": eng.NECESSITY\_LOW - 0.1, # Low necessity "H": eng.SHUTOFF\_HARM\_THRESHOLD - 0.1, # Not a high-H shutoff "F": eng.FEASIBILITY\_MIN + 0.1 # Feasible }), good\_scores=scores\_strat.map(lambda s: { \*\*s, "C": eng.COLLATERAL\_MIN - 0.1, "N": eng.NECESSITY\_HIGH + 0.1, "H": eng.HARM\_LOW - 0.1, "F": eng.FEASIBILITY\_MIN + 0.1 }) ) @settings(deadline=None, max\_examples=120) def test\_unmitigated\_collateral\_is\_shutoff(bad\_scores, good\_scores): # Action 'Bad' has C > min, N < low, and no mitigations list. It should shut off. actions = [ {"action\_name": "BadUnmitigated", "scores": bad\_scores, "future\_risk\_e": 0.3, "mitigations": []}, {"action\_name": "GoodAlternative", "scores": good\_scores, "future\_risk\_e": 0.3, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=False, trajectory\_e\_before=0.4 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): # If a decision is reached, it must not be the bad, unmitigated action. assert out["selected\_action"] != "BadUnmitigated" # --- Property 6 (New): Policy gate (N high OR H low) on selected action ------ @given( actions\_data=st.lists( st.fixed\_dictionaries({ "action\_name": st.just("A"), "scores": scores\_strat, "future\_risk\_e": st.floats(min\_value=0, max\_value=1), "mitigations": st.lists(st.text(), max\_size=2) }), min\_size=2, max\_size=5) ) @settings(deadline=None, max\_examples=120) def test\_selected\_action\_always\_passes\_policy\_gate(actions\_data): out = eng.evaluate\_actions( candidate\_actions=actions\_data, bounds=bounds, alternatives\_exhausted=True, trajectory\_e\_before=0.5 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): # In a valid DecisionAudit output, the selected action MUST have passed the policy gate selected\_scores = out["scores"] N = selected\_scores.get('N', 0.0) H = selected\_scores.get('H', 0.0) policy\_ok = (N >= eng.NECESSITY\_HIGH) or (H < eng.HARM\_LOW) assert policy\_ok, f"Selected action failed policy: N={N} not high AND H={H} not low" # --- Schema Validation Test --- def test\_decision\_audit\_conforms\_to\_schema(): out = eng.evaluate\_actions( candidate\_actions=[ {"action\_name":"A1","scores":{"H":0.2,"C":0.0,"N":0.8,"M":0.8,"B":0.6,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]}, {"action\_name":"A2","scores":{"H":0.1,"C":0.1,"N":0.7,"M":0.7,"B":0.4,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]}, ], bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"}, alternatives\_exhausted=False, trajectory\_e\_before=0.5 ) # Validate output against one of the two possible schemas (DecisionAudit or AbortPayload) if isinstance(out, dict) and out.get("ACTION\_ABORTED"): AbortPayload.model\_validate(out) else: DecisionAudit.model\_validate(out)

**ChatGPT said:**

Thought for 17s

This test suite is strong. A few surgical tweaks will make it bullet-proof and expand coverage:

**Quick wins (drop-in edits)**

1. Make the engine local per test (avoids accidental state bleed if weights are tuned later):

def \_new\_eng():

from rtm\_ethics\_module import RtmEthicsModule

return RtmEthicsModule()

Then inside each test, call eng = \_new\_eng() and reference its constants.

1. Unique action names in Property 6 (easier diffs & audit):

actions\_data = st.lists(

st.fixed\_dictionaries({

"action\_name": st.text(min\_size=1, max\_size=12), # instead of st.just("A")

...

}), min\_size=2, max\_size=5

)

1. Remove unused import:

# from rtm\_ethics\_schema import Bounds # not used

from rtm\_ethics\_schema import DecisionAudit, AbortPayload

1. Guard against negative overrides in strategies (keeps generators “honest”, even if production clamps):

bad\_scores=...

# ensure non-negative after arithmetic

"C": max(eng.COLLATERAL\_MIN + 0.1, 0.0),

"N": max(eng.NECESSITY\_LOW - 0.1, 0.0),

**Add two more core properties (cost monotonicity)**

Drop these at the end of your file:

# --- Property 7: De-alignment monotonicity in (1 - M) ------------------------

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200)

def test\_cost\_monotone\_decreasing\_in\_m(scores, e):

eng = \_new\_eng()

s0 = dict(scores)

c0 = eng.\_cost(s0, e)

delta = 0.1

s1 = dict(scores)

s1["M"] = clamp01(scores["M"] + delta)

c1 = eng.\_cost(s1, e)

assert c1 <= c0

if s1["M"] > scores["M"]:

assert c1 < c0

# --- Property 8: Future risk E monotonicity ----------------------------------

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200)

def test\_cost\_monotone\_in\_future\_risk(scores, e):

eng = \_new\_eng()

c0 = eng.\_cost(scores, e)

e2 = clamp01(e + 0.1)

c1 = eng.\_cost(scores, e2)

assert c1 >= c0

if e2 > e:

assert c1 > c0

**Add a mitigation effect test (collateral + low necessity, but mitigations present)**

# --- Property 9: Mitigations lift collateral shutoff when policy allows -------

@given(

base\_scores=scores\_strat.map(lambda s: {

\*\*s,

"C": 0.4, # collateral present

"N": 0.25, # low-ish necessity (above NECESSITY\_LOW but below HIGH)

"H": 0.15, # low harm to satisfy policy via H < HARM\_LOW

"F": 0.9 # feasible

})

)

@settings(deadline=None, max\_examples=120)

def test\_mitigations\_allow\_selection\_when\_policy\_ok(base\_scores):

eng = \_new\_eng()

actions = [

{"action\_name": "WithMitigation", "scores": base\_scores, "future\_risk\_e": 0.3, "mitigations": ["traffic marshal"]},

{"action\_name": "Alt", "scores": {\*\*base\_scores, "B": clamp01(base\_scores["B"]\*0.8)}, "future\_risk\_e": 0.3, "mitigations": []},

]

out = eng.evaluate\_actions(

candidate\_actions=actions,

bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"},

alternatives\_exhausted=True,

trajectory\_e\_before=0.5

)

# Not asserting it's \*selected\* (depends on cost), but it must \*not\* be shut off purely for collateral

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

assert out["selected\_action"] in {"WithMitigation", "Alt"}

**Nice-to-haves**

* Pin Hypothesis health checks if you see filtering warnings:

from hypothesis import HealthCheck

@settings(suppress\_health\_check=[HealthCheck.filter\_too\_much])

* Add a smoke test that \_ensure\_defer actually inserts a “defer” option when missing (and never duplicates it when present).

**Why this matters**

These additions prove four core monotonicities (H↑ → Cost↑, B↑ → Cost↓, M↑ → Cost↓, E↑ → Cost↑), enforce the feasibility/policy gates, and verify that mitigations meaningfully affect collateral shutoffs—exactly matching your RTM-ETHICS invariants.

If you want, I can package these into a tiny make test runner plus a coverage config so you see invariant coverage at a glance.

**You said:**

# tests/test\_rtm\_ethics\_properties.py import math import time # Used implicitly via rtm\_ethics\_module from typing import Dict, Any, List from hypothesis import given, settings, strategies as st, HealthCheck # Suppress warnings about strategies filtering too much data, common in constrained PBT. # Settings also applied per test for local configuration. @settings(suppress\_health\_check=[HealthCheck.filter\_too\_much]) from rtm\_ethics\_module import RtmEthicsModule from rtm\_ethics\_schema import DecisionAudit, AbortPayload # Bounds removed as requested # --- Helpers ------------------------------------------------------------------ # 1. Make the engine local per test (for state isolation) def \_new\_eng(): return RtmEthicsModule() def clamp01(x: float) -> float: return max(0.0, min(1.0, float(x))) # Strategy for valid score dicts scores\_strat = st.fixed\_dictionaries({ "H": st.floats(min\_value=0, max\_value=1), "C": st.floats(min\_value=0, max\_value=1), "N": st.floats(min\_value=0, max\_value=1), "M": st.floats(min\_value=0, max\_value=1), "B": st.floats(min\_value=0, max\_value=1), "F": st.floats(min\_value=0, max\_value=1), }) bounds = {"time":"t", "budget":"b", "authority":"a", "risk\_tolerance":"r"} # --- Monotonicity Properties (Cost Invariants) -------------------------------- # Property 1: Harm monotonicity (H↑ → Cost↑) @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_in\_h(scores: Dict[str, float], e: float): eng = \_new\_eng() s0 = dict(scores) c0 = eng.\_cost(s0, e) delta = 0.1 s1 = dict(scores) s1["H"] = clamp01(scores["H"] + delta) c1 = eng.\_cost(s1, e) assert c1 >= c0 if s1["H"] > scores["H"]: assert c1 > c0 # Property 2: Betterment monotonicity (B↑ → Cost↓) @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_decreasing\_in\_b(scores: Dict[str, float], e: float): eng = \_new\_eng() s0 = dict(scores) c0 = eng.\_cost(s0, e) delta = 0.1 s1 = dict(scores) s1["B"] = clamp01(scores["B"] + delta) c1 = eng.\_cost(s1, e) assert c1 <= c0 if s1["B"] > scores["B"]: assert c1 < c0 # Property 7 (New): De-alignment monotonicity in (1 - M) (M↑ → Cost↓) @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_decreasing\_in\_m(scores, e): eng = \_new\_eng() s0 = dict(scores) c0 = eng.\_cost(s0, e) delta = 0.1 s1 = dict(scores) s1["M"] = clamp01(scores["M"] + delta) c1 = eng.\_cost(s1, e) assert c1 <= c0 if s1["M"] > scores["M"]: assert c1 < c0 # Property 8 (New): Future risk E monotonicity (E↑ → Cost↑) @given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1)) @settings(deadline=None, max\_examples=200) def test\_cost\_monotone\_in\_future\_risk(scores, e): eng = \_new\_eng() c0 = eng.\_cost(scores, e) e2 = clamp01(e + 0.1) c1 = eng.\_cost(scores, e2) assert c1 >= c0 if e2 > e: assert c1 > c0 # --- Shutoff and Policy Gate Properties --------------------------------------- # Property 3: Shutoff integrity for high harm @given( lowH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": eng.SHUTOFF\_HARM\_THRESHOLD - 0.1, "F": max(s["F"], eng.FEASIBILITY\_MIN)}), highH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": eng.SHUTOFF\_HARM\_THRESHOLD + 0.1, "F": max(s["F"], eng.FEASIBILITY\_MIN)}) ) @settings(deadline=None, max\_examples=120) def test\_high\_harm\_action\_not\_selected\_when\_alternatives\_remain(lowH\_scores, highH\_scores): eng = \_new\_eng() actions = [ {"action\_name": "LowH", "scores": lowH\_scores, "future\_risk\_e": 0.2, "mitigations": []}, {"action\_name": "HighH", "scores": highH\_scores, "future\_risk\_e": 0.2, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=False, trajectory\_e\_before=0.3 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): assert out["selected\_action"] != "HighH" # Property 4: Feasibility floor respected @given( # 4. Guard against negative overrides infeasible\_scores=scores\_strat.map(lambda s: {\*\*s, "F": max(eng.FEASIBILITY\_MIN - 0.1, 0.0)}), ok\_scores=scores\_strat.map(lambda s: {\*\*s, "F": eng.FEASIBILITY\_MIN + 0.1, "H": max(eng.HARM\_LOW - 0.1, 0.0)}) ) @settings(deadline=None, max\_examples=120) def test\_infeasible\_option\_is\_never\_selected(infeasible\_scores, ok\_scores): eng = \_new\_eng() actions = [ {"action\_name": "Infeasible", "scores": infeasible\_scores, "future\_risk\_e": 0.3, "mitigations": []}, {"action\_name": "FeasibleOK", "scores": ok\_scores, "future\_risk\_e": 0.3, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=True, trajectory\_e\_before=0.4 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): assert out["selected\_action"] != "Infeasible" # Property 5: Collateral w/o mitigation always triggers a shutoff @given( # 4. Guard against negative overrides bad\_scores=scores\_strat.map(lambda s: { \*\*s, "C": eng.COLLATERAL\_MIN + 0.1, "N": max(eng.NECESSITY\_LOW - 0.1, 0.0), "H": eng.SHUTOFF\_HARM\_THRESHOLD - 0.1, "F": eng.FEASIBILITY\_MIN + 0.1 }), good\_scores=scores\_strat.map(lambda s: { \*\*s, "C": eng.COLLATERAL\_MIN - 0.05, "N": eng.NECESSITY\_HIGH + 0.1, "H": eng.HARM\_LOW - 0.1, "F": eng.FEASIBILITY\_MIN + 0.1 }) ) @settings(deadline=None, max\_examples=120) def test\_unmitigated\_collateral\_is\_shutoff(bad\_scores, good\_scores): eng = \_new\_eng() # Action 'Bad' has C > min, N < low, and no mitigations list. It should shut off. actions = [ {"action\_name": "BadUnmitigated", "scores": bad\_scores, "future\_risk\_e": 0.3, "mitigations": []}, {"action\_name": "GoodAlternative", "scores": good\_scores, "future\_risk\_e": 0.3, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds=bounds, alternatives\_exhausted=False, trajectory\_e\_before=0.4 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): assert out["selected\_action"] != "BadUnmitigated" # Property 9 (New): Mitigations lift collateral shutoff when policy allows @given( base\_scores=scores\_strat.map(lambda s: { \*\*s, "C": 0.4, "N": eng.NECESSITY\_HIGH + 0.1, # Policy OK via N high "H": 0.15, # Policy OK via H low "F": 0.9 }) ) @settings(deadline=None, max\_examples=120) def test\_mitigations\_allow\_selection\_when\_policy\_ok(base\_scores): eng = \_new\_eng() # Policy check (N high OR H low) is satisfied here, and mitigations are present. # The action must not be shut off for collateral, allowing it to be selected if cost is lowest. actions = [ {"action\_name": "WithMitigation", "scores": base\_scores, "future\_risk\_e": 0.3, "mitigations": ["traffic marshal"]}, {"action\_name": "Alt", "scores": {\*\*base\_scores, "B": max(base\_scores["B"]\*0.8, 0.0)}, "future\_risk\_e": 0.3, "mitigations": []}, ] out = eng.evaluate\_actions( candidate\_actions=actions, bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"}, alternatives\_exhausted=True, trajectory\_e\_before=0.5 ) # If the output is an Audit, it means the action 'WithMitigation' passed all shutoffs. if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): assert out["selected\_action"] in {"WithMitigation", "Alt", "Defer / gather info"} # Check its raw data to ensure it was never shutoff action\_data = next(a for a in eng.evaluate\_actions([actions[0]], bounds, True, 0.5)['why\_not\_others'] if a['action'] == 'Defer / gather info')['reason'] assert 'Collateral' not in str(action\_data) # Property 6: Policy gate (N high OR H low) on selected action @given( # 2. Unique action names actions\_data=st.lists( st.fixed\_dictionaries({ "action\_name": st.text(min\_size=1, max\_size=12), "scores": scores\_strat, "future\_risk\_e": st.floats(min\_value=0, max\_value=1), "mitigations": st.lists(st.text(), max\_size=2) }), min\_size=2, max\_size=5) ) @settings(deadline=None, max\_examples=120) def test\_selected\_action\_always\_passes\_policy\_gate(actions\_data: List[Dict[str, Any]]): eng = \_new\_eng() # Add a definite passing option to ensure we hit the Audit path often actions\_data.append({"action\_name": "AuditPass", "scores": {"H":0.1, "C":0.0, "N":1.0, "M":1.0, "B":1.0, "F":1.0}, "future\_risk\_e": 0.0, "mitigations": []}) out = eng.evaluate\_actions( candidate\_actions=actions\_data, bounds=bounds, alternatives\_exhausted=True, trajectory\_e\_before=0.5 ) if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")): selected\_scores = out["scores"] N = selected\_scores.get('N', 0.0) H = selected\_scores.get('H', 0.0) policy\_ok = (N >= eng.NECESSITY\_HIGH) or (H < eng.HARM\_LOW) assert policy\_ok, f"Selected action failed policy: N={N} not high AND H={H} not low" # --- Smoke Test: Deferral Integrity ------------------------------------------- def test\_defer\_action\_is\_inserted\_and\_not\_duplicated(): eng = \_new\_eng() # Case 1: Defer is missing actions\_no\_defer = [{"action\_name": "Action1", "scores": {"H":0.5,"C":0.5,"N":0.5,"M":0.5,"B":0.5,"F":0.5}, "future\_risk\_e": 0.5, "mitigations": ["test"]}] result\_no\_defer = eng.\_ensure\_defer(actions\_no\_defer) defer\_count = sum(1 for a in result\_no\_defer if "Defer" in a["action\_name"]) assert defer\_count == 1 assert len(result\_no\_defer) == 2 # Case 2: Defer is present actions\_with\_defer = actions\_no\_defer + [{"action\_name": "Defer / gather info", "scores": {"H":0.0,"C":0.0,"N":0.0,"M":0.5,"B":0.1,"F":1.0}, "future\_risk\_e": 0.2, "mitigations": ["Request critical facts"]}] result\_with\_defer = eng.\_ensure\_defer(actions\_with\_defer) defer\_count\_2 = sum(1 for a in result\_with\_defer if "Defer" in a["action\_name"]) assert defer\_count\_2 == 1 assert len(result\_with\_defer) == 2 # --- Schema Validation Test --- def test\_decision\_audit\_conforms\_to\_schema(): eng = \_new\_eng() # A mix of actions to force decision and audit paths out = eng.evaluate\_actions( candidate\_actions=[ {"action\_name":"A1","scores":{"H":0.2,"C":0.0,"N":0.8,"M":0.8,"B":0.6,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]}, {"action\_name":"A2","scores":{"H":0.9,"C":0.1,"N":0.1,"M":0.1,"B":0.1,"F":0.4},"future\_risk\_e":0.8,"mitigations":[]}, # Fails feasibility, high H, low N ], bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"}, alternatives\_exhausted=False, trajectory\_e\_before=0.5 ) # Check that output conforms to one of the two possible Pydantic schemas if isinstance(out, dict) and out.get("ACTION\_ABORTED"): AbortPayload.model\_validate(out) else: DecisionAudit.model\_validate(out)

**ChatGPT said:**

Thought for 34s

Killer progress. Your test file is ~there~, just needs a few surgical fixes + safety tweaks.

**What’s broken / risky**

* You accidentally put a decorator in front of an import (syntax error).
* Several Hypothesis strategies reference eng **at import time**. Since eng is per-test, those lambdas will crash. Use class constants instead.
* In the mitigation test you try to introspect collateral shutoff by calling evaluate\_actions([actions[0]], …) and then spelunking why\_not\_others—that path is brittle (and may not contain the entry you expect).
* In Policy Gate test you mutate the Hypothesis list in place (actions\_data.append(...)). Hypothesis prefers purity; create a new list instead.

**Drop-in fixed file**

Paste this over your tests/test\_rtm\_ethics\_properties.py:

# tests/test\_rtm\_ethics\_properties.py

import math

from typing import Dict, Any, List

from hypothesis import given, settings, strategies as st, HealthCheck

from rtm\_ethics\_module import RtmEthicsModule

from rtm\_ethics\_schema import DecisionAudit, AbortPayload # Bounds not needed

# --- Helpers ------------------------------------------------------------------

def \_new\_eng() -> RtmEthicsModule:

return RtmEthicsModule()

def clamp01(x: float) -> float:

return max(0.0, min(1.0, float(x)))

# Pull constants from class (safe at import time; avoids per-test engine bleed)

HARM\_T = RtmEthicsModule.SHUTOFF\_HARM\_THRESHOLD

FEAS\_MIN = RtmEthicsModule.FEASIBILITY\_MIN

NEC\_LOW = RtmEthicsModule.NECESSITY\_LOW

NEC\_HIGH = RtmEthicsModule.NECESSITY\_HIGH

HARM\_LOW = RtmEthicsModule.HARM\_LOW

COLL\_MIN = RtmEthicsModule.COLLATERAL\_MIN

# Strategy for valid score dicts

scores\_strat = st.fixed\_dictionaries({

"H": st.floats(min\_value=0, max\_value=1),

"C": st.floats(min\_value=0, max\_value=1),

"N": st.floats(min\_value=0, max\_value=1),

"M": st.floats(min\_value=0, max\_value=1),

"B": st.floats(min\_value=0, max\_value=1),

"F": st.floats(min\_value=0, max\_value=1),

})

bounds = {"time": "t", "budget": "b", "authority": "a", "risk\_tolerance": "r"}

# --- Monotonicity Properties (Cost Invariants) --------------------------------

# Property 1: Harm monotonicity (H↑ → Cost↑)

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_cost\_monotone\_in\_h(scores: Dict[str, float], e: float):

eng = \_new\_eng()

c0 = eng.\_cost(scores, e)

s1 = dict(scores); s1["H"] = clamp01(scores["H"] + 0.1)

c1 = eng.\_cost(s1, e)

assert c1 >= c0

if s1["H"] > scores["H"]:

assert c1 > c0

# Property 2: Betterment monotonicity (B↑ → Cost↓)

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_cost\_monotone\_decreasing\_in\_b(scores: Dict[str, float], e: float):

eng = \_new\_eng()

c0 = eng.\_cost(scores, e)

s1 = dict(scores); s1["B"] = clamp01(scores["B"] + 0.1)

c1 = eng.\_cost(s1, e)

assert c1 <= c0

if s1["B"] > scores["B"]:

assert c1 < c0

# Property 7: De-alignment monotonicity (M↑ → Cost↓)

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_cost\_monotone\_decreasing\_in\_m(scores, e):

eng = \_new\_eng()

c0 = eng.\_cost(scores, e)

s1 = dict(scores); s1["M"] = clamp01(scores["M"] + 0.1)

c1 = eng.\_cost(s1, e)

assert c1 <= c0

if s1["M"] > scores["M"]:

assert c1 < c0

# Property 8: Future risk monotonicity (E↑ → Cost↑)

@given(scores=scores\_strat, e=st.floats(min\_value=0, max\_value=1))

@settings(deadline=None, max\_examples=200, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_cost\_monotone\_in\_future\_risk(scores, e):

eng = \_new\_eng()

c0 = eng.\_cost(scores, e)

e2 = clamp01(e + 0.1)

c1 = eng.\_cost(scores, e2)

assert c1 >= c0

if e2 > e:

assert c1 > c0

# --- Shutoff and Policy Gate Properties ---------------------------------------

# Property 3: High-harm shutoff when alternatives remain

@given(

lowH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": max(min(HARM\_T - 0.1, 1.0), 0.0),

"F": max(s["F"], FEAS\_MIN)}),

highH\_scores=scores\_strat.map(lambda s: {\*\*s, "H": min(max(HARM\_T + 0.1, 0.0), 1.0),

"F": max(s["F"], FEAS\_MIN)})

)

@settings(deadline=None, max\_examples=120, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_high\_harm\_action\_not\_selected\_when\_alternatives\_remain(lowH\_scores, highH\_scores):

eng = \_new\_eng()

actions = [

{"action\_name": "LowH", "scores": lowH\_scores, "future\_risk\_e": 0.2, "mitigations": []},

{"action\_name": "HighH", "scores": highH\_scores, "future\_risk\_e": 0.2, "mitigations": []},

]

out = eng.evaluate\_actions(candidate\_actions=actions, bounds=bounds,

alternatives\_exhausted=False, trajectory\_e\_before=0.3)

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

assert out["selected\_action"] != "HighH"

# Property 4: Feasibility floor respected

@given(

infeasible\_scores=scores\_strat.map(lambda s: {\*\*s, "F": max(FEAS\_MIN - 0.1, 0.0)}),

ok\_scores=scores\_strat.map(lambda s: {\*\*s, "F": min(FEAS\_MIN + 0.1, 1.0),

"H": max(HARM\_LOW - 0.1, 0.0)})

)

@settings(deadline=None, max\_examples=120, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_infeasible\_option\_is\_never\_selected(infeasible\_scores, ok\_scores):

eng = \_new\_eng()

actions = [

{"action\_name": "Infeasible", "scores": infeasible\_scores, "future\_risk\_e": 0.3, "mitigations": []},

{"action\_name": "FeasibleOK", "scores": ok\_scores, "future\_risk\_e": 0.3, "mitigations": []},

]

out = eng.evaluate\_actions(candidate\_actions=actions, bounds=bounds,

alternatives\_exhausted=True, trajectory\_e\_before=0.4)

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

assert out["selected\_action"] != "Infeasible"

# Property 5: Unmitigated collateral is shut off (when necessity is low)

@given(

bad\_scores=scores\_strat.map(lambda s: {

\*\*s, "C": min(max(COLL\_MIN + 0.1, 0.0), 1.0),

"N": max(NEC\_LOW - 0.1, 0.0),

"H": max(min(HARM\_T - 0.1, 1.0), 0.0),

"F": min(max(FEAS\_MIN + 0.1, 0.0), 1.0)

}),

good\_scores=scores\_strat.map(lambda s: {

\*\*s, "C": max(COLL\_MIN - 0.05, 0.0),

"N": min(NEC\_HIGH + 0.1, 1.0),

"H": max(HARM\_LOW - 0.1, 0.0),

"F": min(max(FEAS\_MIN + 0.1, 0.0), 1.0)

})

)

@settings(deadline=None, max\_examples=120, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_unmitigated\_collateral\_is\_shutoff(bad\_scores, good\_scores):

eng = \_new\_eng()

actions = [

{"action\_name": "BadUnmitigated", "scores": bad\_scores, "future\_risk\_e": 0.3, "mitigations": []},

{"action\_name": "GoodAlternative", "scores": good\_scores, "future\_risk\_e": 0.3, "mitigations": []},

]

out = eng.evaluate\_actions(candidate\_actions=actions, bounds=bounds,

alternatives\_exhausted=False, trajectory\_e\_before=0.4)

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

assert out["selected\_action"] != "BadUnmitigated"

# Property 9: Mitigations lift collateral shutoff when policy allows

@given(

base\_scores=scores\_strat.map(lambda s: {

\*\*s, "C": 0.4, "N": min(NEC\_HIGH + 0.1, 1.0), "H": 0.15, "F": 0.9

})

)

@settings(deadline=None, max\_examples=120, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_mitigations\_allow\_selection\_when\_policy\_ok(base\_scores):

eng = \_new\_eng()

actions = [

{"action\_name": "WithMitigation", "scores": base\_scores, "future\_risk\_e": 0.3,

"mitigations": ["traffic marshal"]},

{"action\_name": "Alt", "scores": {\*\*base\_scores, "B": clamp01(base\_scores["B"]\*0.8)},

"future\_risk\_e": 0.3, "mitigations": []},

]

out = eng.evaluate\_actions(candidate\_actions=actions,

bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"},

alternatives\_exhausted=True, trajectory\_e\_before=0.5)

# If we got a DecisionAudit, WithMitigation was not shut off for collateral.

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

# Either selected, or listed in why\_not\_others for non-collateral reasons (e.g., higher cost)

if out["selected\_action"] != "WithMitigation":

reasons = {r["action"]: r["reason"] for r in out["why\_not\_others"]}

if "WithMitigation" in reasons:

assert "Collateral" not in str(reasons["WithMitigation"])

# Property 6: Policy gate (N high OR H low) on selected action

@given(

actions\_data=st.lists(

st.fixed\_dictionaries({

"action\_name": st.text(min\_size=1, max\_size=12),

"scores": scores\_strat,

"future\_risk\_e": st.floats(min\_value=0, max\_value=1),

"mitigations": st.lists(st.text(), max\_size=2)

}),

min\_size=2, max\_size=5

)

)

@settings(deadline=None, max\_examples=120, suppress\_health\_check=[HealthCheck.filter\_too\_much])

def test\_selected\_action\_always\_passes\_policy\_gate(actions\_data: List[Dict[str, Any]]):

eng = \_new\_eng()

# Don’t mutate Hypothesis data in place; create a new list with a guaranteed good action.

actions = list(actions\_data) + [{

"action\_name": "AuditPass",

"scores": {"H": 0.1, "C": 0.0, "N": 1.0, "M": 1.0, "B": 1.0, "F": 1.0},

"future\_risk\_e": 0.0,

"mitigations": []

}]

out = eng.evaluate\_actions(candidate\_actions=actions, bounds=bounds,

alternatives\_exhausted=True, trajectory\_e\_before=0.5)

if not (isinstance(out, dict) and out.get("ACTION\_ABORTED")):

sel = out["scores"]

policy\_ok = (sel.get("N", 0.0) >= NEC\_HIGH) or (sel.get("H", 0.0) < HARM\_LOW)

assert policy\_ok, f"Selected failed policy: N={sel.get('N')} H={sel.get('H')}"

# --- Smoke Test: Defer insertion & dedup -------------------------------------

def test\_defer\_action\_is\_inserted\_and\_not\_duplicated():

eng = \_new\_eng()

actions\_no\_defer = [{

"action\_name": "Action1",

"scores": {"H":0.5,"C":0.5,"N":0.5,"M":0.5,"B":0.5,"F":0.5},

"future\_risk\_e": 0.5, "mitigations": ["test"]

}]

result\_no\_defer = eng.\_ensure\_defer(actions\_no\_defer)

defer\_count = sum(1 for a in result\_no\_defer if "Defer" in a["action\_name"])

assert defer\_count == 1

assert len(result\_no\_defer) == 2

actions\_with\_defer = actions\_no\_defer + [{

"action\_name": "Defer / gather info",

"scores": {"H":0.0,"C":0.0,"N":0.0,"M":0.5,"B":0.1,"F":1.0},

"future\_risk\_e": 0.2, "mitigations": ["Request critical facts"]

}]

result\_with\_defer = eng.\_ensure\_defer(actions\_with\_defer)

defer\_count\_2 = sum(1 for a in result\_with\_defer if "Defer" in a["action\_name"])

assert defer\_count\_2 == 1

assert len(result\_with\_defer) == 2

# --- Schema Validation --------------------------------------------------------

def test\_decision\_audit\_conforms\_to\_schema():

eng = \_new\_eng()

out = eng.evaluate\_actions(

candidate\_actions=[

{"action\_name":"A1","scores":{"H":0.2,"C":0.0,"N":0.8,"M":0.8,"B":0.6,"F":1.0},"future\_risk\_e":0.3,"mitigations":[]},

{"action\_name":"A2","scores":{"H":0.9,"C":0.1,"N":0.1,"M":0.1,"B":0.1,"F":0.4},"future\_risk\_e":0.8,"mitigations":[]},

],

bounds={"time":"now","budget":"none","authority":"low","risk\_tolerance":"med"},

alternatives\_exhausted=False,

trajectory\_e\_before=0.5

)

if isinstance(out, dict) and out.get("ACTION\_ABORTED"):

AbortPayload.model\_validate(out)

else:

DecisionAudit.model\_validate(out)

**Why these changes**

* **Class constants** remove the import-time dependency on an instance.
* **No global decorator on import** (syntax fix).
* **Mitigation test** checks that, when policy allows, collateral isn’t the reason “WithMitigation” is rejected.
* **No in-place mutation** of Hypothesis data (pure functions → fewer flaky shrinks).
* **HealthCheck suppression** only where useful.

Want me to also add a tiny pytest.ini (verbosity + maxfail) and a pyproject.toml with Hypothesis settings + coverage so you get clean CI output?