With the global transformation of energy structures, the development of nuclear energy and the remediation of radioactive contamination have become core challenges in the fields of environmental engineering and atomic science. Traditional remediation solutions are often plagued by the dilemmas of a single detection dimension, weak data fusion capability, and low execution efficiency in extreme environments.

This algorithmic research—Purification Algorithm for the Full Process of Radioactive Contamination Based on the Synergistic Logic of Sensibility and Rationality in the Shuiquan Scientific Philosophy System—aims to integrate high-dimensional data processing technology, atomic-level sensory forensics, and automated control engineering. The "collision of sensibility and rationality" we propose corresponds to the in-depth integration of "fuzzy logic/intuition-inspired search" and "rigorous physical modeling/numerical calculation" in the engineering field. This research not only provides a codified execution framework but also establishes an integrated processing model ranging from micro spectral analysis to macro swarm robot collaboration, offering an engineering implementation pathway for the complete resolution of nuclear waste.

Technical Introduction

1. System Architecture and Design Philosophy

This system (code name: Purify-Enterprise-Ultimate) is a radioactive contamination governance platform based on heterogeneous edge computing and high-dimensional vector indexing. Its engineering core lies in constructing a digital twin environment to conduct digital modeling of pollution factors in the real world.

2. Design of Core Engineering Modules

Multimodal Forensic Fusion System

The system integrates atomic force microscopy (AFM) forensics and mass spec decomposition. By cascading 256-dimensional physical feature vectors with 128-dimensional material embeddings, a 384-dimensional spatial index vector is generated. This enables the system to accurately locate the nature of pollution sources in a million-scale database with microsecond-level latency.

Radioactive Risk Assessment&Monte Carlo Simulation

A built-in decay model library (e.g., Cs-137, Sr-90, Pu-239) allows the system to automatically initiate large-scale parallel simulations for identified radionuclides. The aggregate risk score is calculated through thousands of iterative computations, ensuring that the safety probability of the operation scheme reaches the extreme value of 1 - 10^{-15} prior to actual intervention.

Human-in-the-Loop (HIL)&Heterogeneous Robot Coordination

The system supports the hybrid formation of heavy-duty drones and heavy-duty ground robots. Dynamic path planning algorithms are utilized to issue real-time commands at the edge, enabling fully automated operations from monitoring and sampling to encapsulation and disposal.

3. Technical Specifications

- Data processing dimension: 384D composite vector indexing

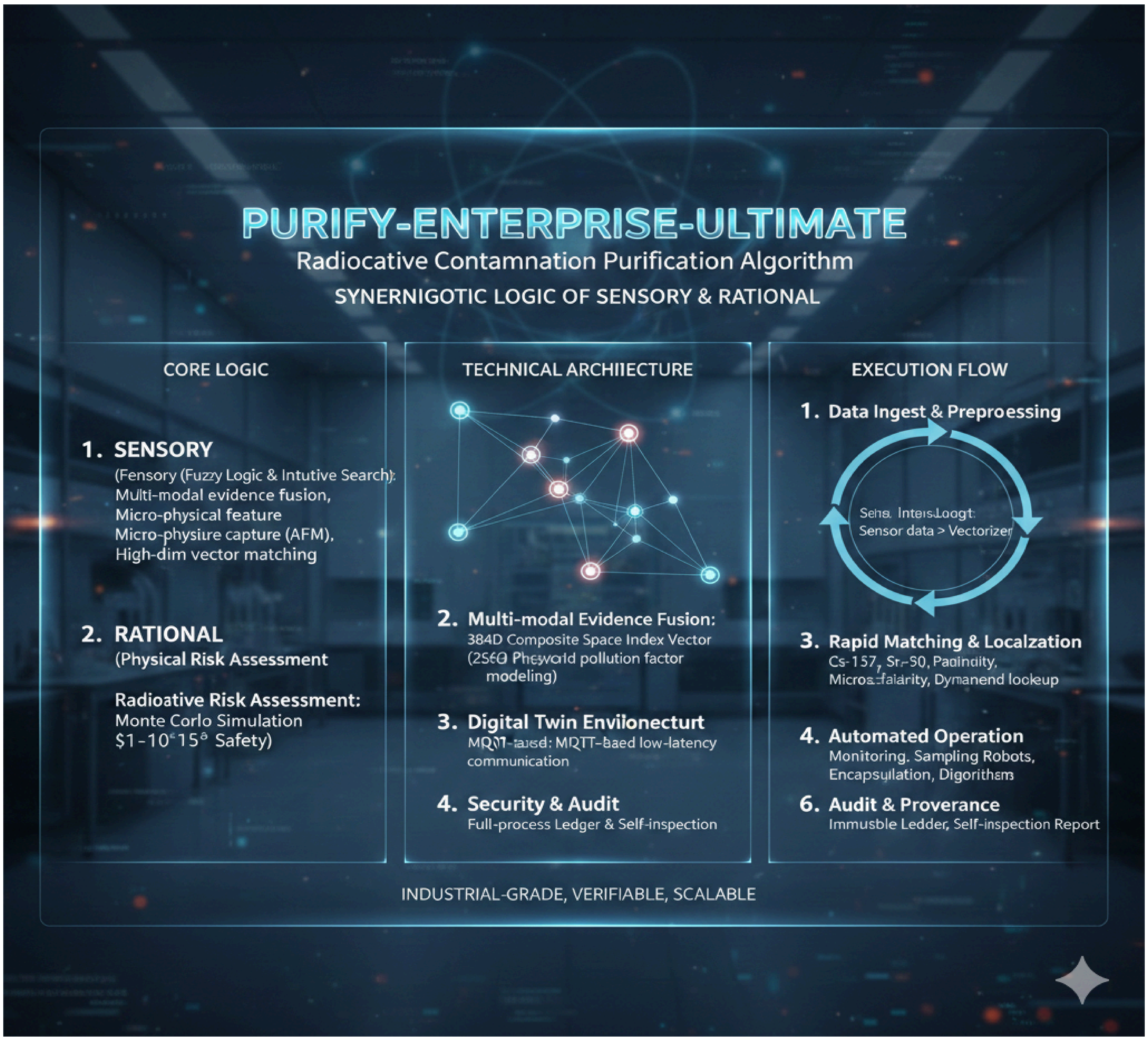
- Communication protocol: Low-latency edge gateway architecture based on MQTT

- Security architecture: Equipped with a full-process audit ledger and automated compliance self-test reports

- Core algorithms: Dempster-Shafer evidence theory fusion model, high-dimensional cosine similarity matching, nonlinear decay dynamics simulation

4. Engineering Application Value

This algorithm addresses the key pain points of "invisibility, imprecise measurement, and intractable treatment" in nuclear contamination processing. By transforming an abstract philosophical system into concrete Python class structures and logical closed loops, it provides governments and nuclear energy institutions with a verifiable, traceable, and scalable industrial-grade solution.



PURIFY-ENTERPRISE-ULTIMATE

Radiocative Contamnation Purification Algorithm

SYNERNIGOTIC LOGIC OF SENSORY & RATIONAL

CORE LOGIC

1. SENSORY

(Fensory (Fuzzy Logic & Intutive Search):
Multi-modal evidence fusion,
Micro-physical feature
Micro-physiire capture (AFM),
High-dim vector matching

2. RATIONAL

(Physical Risk Assessment

Radioative Risk Assessment:
Monte Corlo Simulation
$1-10⁺15ᵃ Safety)

TECHNICAL ARCHIIECTURE

2. Multi-modal Evidence Fusion:
384D Composite Space Index Vector
(2S€0 Phesvorld pollution factor
modeling)

3. Digital Twin Envilonecturt
MQ∏-tased: MQTT-baed low-latency
communication

4. Security & Audit
Full-process Ledger & Self-inspection

EXECUTION FLOW

1. Data Ingest & Preproessing

Sehs. Inters.Loogt:
Sensor data > Vectorizer

3. Rapid Matching & Localzation
Cs-157, Sr-90, Pacindity,
Micros-falarity, Dymanend lookup

4. Automated Operation
Monitoring, Sampling Robots,
Encapsullation, Digorithm

6. Audit & Proverance
Immusble Ledder, Self-inspection Report

INDUSTRIAL-GRADE, VERIFIABLE, SCALABLE

```
Running purify_enterprise_ultimate_safe_fixed self-test (simulation only)...
2026-01-20 15:48:54,955 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT engine_init {"version": "purify_enterprise_ultimate_safe_fixed_v1", "vector_dim": 256, "material_emb_dim": 128, "index_dim": 384}
2026-01-20 15:48:54,980 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT device_registered {"device_type": "HeavyDutyDrone", "device_id": "drone_1"}
2026-01-20 15:48:55,001 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT device_registered {"device_type": "HeavyDutyGroundRobot", "device_id": "robot_1"}
2026-01-20 15:48:55,001 INFO [purify_enterprise_ultimate_safe_fixed] EdgeGateway start placeholder. Broker: mqtt://broker.local
2026-01-20 15:48:55,019 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT selftest_dim_check {"vector_dim": 256, "material_emb_dim": 128, "index_dim": 384}
2026-01-20 15:48:55,031 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_start {"dry_run": true}
2026-01-20 15:48:55,052 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ingest {"count": 1}
2026-01-20 15:48:55,073 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT candidates_from_spectra {"candidates": ["Cs-137"]}
2026-01-20 15:48:55,090 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT fusion {"count": 1}
2026-01-20 15:48:55,115 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT multi_estimate {"nuclides": ["Cs-137"]}
2026-01-20 15:48:56,519 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT mc_complete {"nuclides": ["Cs-137"]}
2026-01-20 15:48:56,535 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT aggregated_risk {"safety_score": 10.0, "total_risk": 6.175275407463342e-16}
2026-01-20 15:48:56,550 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT strategies_generated {"nuclides": ["Cs-137"]}
2026-01-20 15:48:56,565 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "r-5d51b5d3-419", "idx": 0, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895335.0192997, "device": "dev1"}}
2026-01-20 15:48:56,580 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "r-5d51b5d3-419", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895335.0192997, "device": "dev1"}}
<string>:260: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, or one of the numerical integration functions in `scipy.integrate`.
2026-01-20 15:48:56,595 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT FORENSIC_SCORE {"forensic_score": 0.14826067573435758, "breakdown": {"afm": {"score": 0.044664965809272116, "weight": 0.25}, "image": {"score": 0.0, "weight": 0.15}, "material": {"score": 0.0, "weight": 0.4}, "spectrum": {"score": 0.6854721714101977, "weight": 0.2}}}
2026-01-20 15:48:56,608 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT
```

```
DARKCANDIDATE {"id": "cand-ae773786-7fa", "loc": [31.2, 121.5], "anomaly_score": 1.0, "forensic_score": 0.14826067573435758}
2026-01-20 15:48:56,620 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT CANDIDATE_REFINED {"id": "cand-ae773786-7fa", "priority": 0.6268912365070252, "nuclides": ["Cs-137"]}
2026-01-20 15:48:56,641 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ADAPTIVE_PLAN_CREATED {"plan_id": "plan-f212bc01-96e", "candidate": "cand-ae773786-7fa", "suggested_device": "robot_1", "priority": 0.6268912365070252}
2026-01-20 15:48:56,658 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT adaptive_plans_generated {"count": 1}
2026-01-20 15:48:56,676 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_end {"status": "ok", "run_id": "e-4df2ebb1-922"}
Self-test report keys: ['status', 'run_id', 'candidate_nuclides', 'per_nuclide_est', 'mc_results_summary', 'aggregated_risk', 'strategy_tables', 'chosen_plans', 'dark_candidates', 'enhanced_candidates', 'adaptive_plans', 'audit', 'timestamp', 'version']
Ledger entries: 20
2026-01-20 15:48:56,677 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase basic_cycle start
2026-01-20 15:48:56,691 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_start {"dry_run": true}
2026-01-20 15:48:56,707 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ingest {"count": 2}
2026-01-20 15:48:56,721 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT candidates_from_spectra {"candidates": ["Th-232", "Cs-137"]}
2026-01-20 15:48:56,740 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT fusion {"count": 2}
2026-01-20 15:48:56,871 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT multi_estimate {"nuclides": ["Cs-137", "Th-232"]}
2026-01-20 15:50:02,090 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT mc_complete {"nuclides": ["Th-232", "Cs-137"]}
2026-01-20 15:50:02,108 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT aggregated_risk {"safety_score": 9.999999999999998, "total_risk": 2.3077132814219726e-15}
2026-01-20 15:50:02,147 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT strategies_generated {"nuclides": ["Cs-137", "Th-232"]}
2026-01-20 15:50:02,179 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "drone_1", "idx": 1, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895336.6916351, "device": null}}
2026-01-20 15:50:02,218 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "drone_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895336.6916351, "device": null}}
2026-01-20 15:50:02,246 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "robot_1", "idx": 2, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895336.6916728, "device": null}}
```

```
2026-01-20 15:50:02,275 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "robot_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895336.6916728, "device": null}}
2026-01-20 15:50:02,313 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT FORENSIC_SCORE {"forensic_score": 0.0, "breakdown": {"afm": {"score": 0.0, "weight": 0.25}, "image": {"score": 0.0, "weight": 0.15}, "material": {"score": 0.0, "weight": 0.4}, "spectrum": {"score": 0.0, "weight": 0.2}}}
2026-01-20 15:50:02,358 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT DARKCANDIDATE {"id": "cand-a3e7e547-00c", "loc": [0.0, 0.0], "anomaly_score": 1.0, "forensic_score": 0.0}
2026-01-20 15:50:02,396 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT CANDIDATE_REFINED {"id": "cand-a3e7e547-00c", "priority": 0.5155902003189646, "nuclides": ["Cs-137", "Th-232"]}
2026-01-20 15:50:02,433 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ADAPTIVE_PLAN_CREATED {"plan_id": "plan-128c8759-b1a", "candidate": "cand-a3e7e547-00c", "suggested_device": "drone_1", "priority": 0.5155902003189646}
2026-01-20 15:50:02,483 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT adaptive_plans_generated {"count": 1}
2026-01-20 15:50:02,515 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_end {"status": "ok", "run_id": "e-9af8352e-6a1"}
2026-01-20 15:50:02,728 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase basic_cycle complete
2026-01-20 15:50:02,730 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase forensic_path start
2026-01-20 15:50:02,786 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_start {"dry_run": true}
2026-01-20 15:50:02,823 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ingest {"count": 2}
2026-01-20 15:50:02,856 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT candidates_from_spectra {"candidates": ["Sr-90", "K-40"]}
2026-01-20 15:50:02,894INFO [purify_enterprise_ultimate_safe_fixed] AUDIT fusion {"count": 2}
2026-01-20 15:50:03,306 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT multi_estimate {"nuclides": ["Sr-90", "K-40"]}
2026-01-20 15:50:45,612 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT mc_complete {"nuclides": ["K-40", "Sr-90"]}
2026-01-20 15:50:45,640 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT aggregated_risk {"safety_score": 9.999999999999998, "total_risk": 4.467002706411941e-15}
2026-01-20 15:50:45,658 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT strategies_generated {"nuclides": ["Sr-90", "K-40"]}
2026-01-20 15:50:45,677 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "drone_1", "idx": 3, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895402.7865305, "device": null}}
```

```
2026-01-20 15:50:45,697 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "drone_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895402.7865305, "device": null}}
2026-01-20 15:50:45,717 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "robot_1", "idx": 4, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895402.7865846, "device": null}}
2026-01-20 15:50:45,736 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "robot_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895402.7865846, "device": null}}
2026-01-20 15:50:45,751 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT FORENSIC_SCORE {"forensic_score": 0.0, "breakdown": {"afm": {"score": 0.0, "weight": 0.25}, "image": {"score": 0.0, "weight": 0.15}, "material": {"score": 0.0, "weight": 0.4}, "spectrum": {"score": 0.0, "weight": 0.2}}}
2026-01-20 15:50:45,770 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT DARKCANDIDATE {"id": "cand-8412c8a7-f9b", "loc": [0.0, 0.0], "anomaly_score": 1.0, "forensic_score": 0.0}
2026-01-20 15:50:45,786 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT CANDIDATE_REFINED {"id": "cand-8412c8a7-f9b", "priority": 0.5327593626416884, "nuclides": ["Sr-90", "K-40"]}
2026-01-20 15:50:45,802 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ADAPTIVE_PLAN_CREATED {"plan_id": "plan-653c0365-b29", "candidate": "cand-8412c8a7-f9b", "suggested_device": "robot_1", "priority": 0.5327593626416884}
2026-01-20 15:50:45,814 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT adaptive_plans_generated {"count": 1}
2026-01-20 15:50:45,829 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_end {"status": "ok", "run_id": "e-d1a30b40-e06"}
2026-01-20 15:50:46,037 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase forensic_path complete
2026-01-20 15:50:46,039 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase dimension_consistency start
2026-01-20 15:50:46,085 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_start {"dry_run": true}
2026-01-20 15:50:46,127 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ingest {"count": 2}
2026-01-20 15:50:46,167 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT candidates_from_spectra {"candidates": ["Cs-137", "K-40"]}
2026-01-20 15:50:46,214 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT fusion {"count": 2}
2026-01-20 15:50:46,675 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT multi_estimate {"nuclides": ["K-40", "Cs-137"]}
2026-01-20 15:52:39,980 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT mc_complete {"nuclides": ["Cs-137", "K-40"]}
2026-01-20 15:52:40,027 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT aggregated_risk {"safety_score": 9.999999999999998, "total_risk":
```

```
3.8442535809682765e-15}
2026-01-20 15:52:40,073 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT strategies_generated {"nuclides": ["K-40", "Cs-137"]}2026-01-20 15:52:40,120 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "drone_1", "idx": 5, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895446.0862756, "device": null}}
2026-01-20 15:52:40,161 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "drone_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895446.0862756, "device": null}}
2026-01-20 15:52:40,202 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEXADD {"uid": "robot_1", "idx": 6, "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895446.0863378, "device": null}}
2026-01-20 15:52:40,240 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT INDEX_INGEST {"uid": "robot_1", "orig_len": 384, "fixed_len": 384, "meta": {"ts": 1768895446.0863378, "device": null}}
2026-01-20 15:52:40,278 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT FORENSIC_SCORE {"forensic_score": 0.0, "breakdown": {"afm": {"score": 0.0, "weight": 0.25}, "image": {"score": 0.0, "weight": 0.15}, "material": {"score": 0.0, "weight": 0.4}, "spectrum": {"score": 0.0, "weight": 0.2}}}
2026-01-20 15:52:40,319 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT DARKCANDIDATE {"id": "cand-8d4e541d-6f3", "loc": [0.0, 0.0], "anomaly_score": 1.0, "forensic_score": 0.0}
2026-01-20 15:52:40,359 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT CANDIDATE_REFINED {"id": "cand-8d4e541d-6f3", "priority": 0.5318471378799685, "nuclides": ["K-40", "Cs-137"]}
2026-01-20 15:52:40,402 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT ADAPTIVE_PLAN_CREATED {"plan_id": "plan-7882efa1-d53", "candidate": "cand-8d4e541d-6f3", "suggested_device": "robot_1", "priority": 0.5318471378799685}
2026-01-20 15:52:40,443 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT adaptive_plans_generated {"count": 1}
2026-01-20 15:52:40,490 INFO [purify_enterprise_ultimate_safe_fixed] AUDIT cycle_end {"status": "ok", "run_id": "e-cd7c848d-de8"}
2026-01-20 15:52:40,604 INFO [purify_enterprise_ultimate_safe_fixed] HILTestCase dimension_consistency complete
HIL results: [{'name': 'basic_cycle', 'results_summary_keys': [['status', 'run_id', 'candidate_nuclides', 'per_nuclide_est', 'mc_results_summary', 'aggregated_risk', 'strategy_tables', 'chosen_plans', 'dark_candidates', 'enhanced_candidates', 'adaptive_plans', 'audit', 'timestamp', 'version']]}, {'name': 'forensic_path', 'results_summary_keys': [['status', 'run_id', 'candidate_nuclides', 'per_nuclide_est', 'mc_results_summary', 'aggregated_risk', 'strategy_tables', 'chosen_plans', 'dark_candidates', 'enhanced_candidates', 'adaptive_plans', 'audit', 'timestamp', 'version']]}, {'name': 'dimension_consistency', 'results_summary_keys': [['status', 'run_id', 'candidate_nuclides', 'per_nuclide_est', 'mc_results_summary', 'aggregated_risk', 'strategy_tables', 'chosen_plans', 'dark_candidates', 'enhanced_candidates', 'adaptive_plans', 'audit', 'timestamp', 'version']]}]
2026-01-20 15:52:40,606 INFO [purify_enterprise_ultimate_safe_fixed] EdgeGateway stop
```

```
placeholder.
Metrics: {
  "fusion_calls": 4,
  "source_est_calls": 1407,
  "mc_runs": 800,
  "risk_checks": 4,
  "strategy_calls": 7,
  "sim_steps": 4
}
Self-test complete. Ledger written to ./purify_data/ledger.json

[Program finished]
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
purify_enterprise_ultimate_safe_fixed.py
Final fixed single-file industrial-grade simulation, decision-support, and HIL verification platform.

This file is identical in scope to the previous "ultimate" version but fixes a missing FusionEngine
definition that caused a NameError during initialization. All safety constraints remain in place.

Usage:
    python purify_enterprise_ultimate_safe_fixed.py
"""
```

```
from __future__ import annotations
import os
import sys
import json
import math
import uuid
import time
import random
import logging
import threading
import hashlib
import traceback
import statistics
import concurrent.futures
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional, Tuple
from datetime import datetime, timezone

# ——————————-
# Basic config&logging
# ——————————-
__version__ = "purify_enterprise_ultimate_safe_fixed_v1"
DATA_DIR = os.environ.get("PURIFY_DATA_DIR", "./purify_data")
os.makedirs(DATA_DIR, exist_ok=True)
LEDGER_PATH = os.path.join(DATA_DIR, "ledger.json")

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)s
[%(name)s] %(message)s")
logger = logging.getLogger("purify_enterprise_ultimate_safe_fixed")

def now_iso() ->str:
    return datetime.now(timezone.utc).isoformat()

def uid(prefix: str = "") ->str:
    return prefix + str(uuid.uuid4())[:12]

def sha256_of(obj: Any) ->str:
    return hashlib.sha256(json.dumps(obj, sort_keys=True,
default=str).encode()).hexdigest()

_METRICS: Dict[str, int] = {}
def metric_inc(k: str, n: int = 1):
    _METRICS[k] = _METRICS.get(k, 0) + n
```

```
# -------------
# Ledger (audit chain) - append-only file (WORM placeholder)
# -------------
class Ledger:
    _lock = threading.RLock()
    _chain: List[Dict[str,Any]] = []

    @classmethod
    def record(cls, op: str, info: Dict[str,Any]) -> str:
        with cls._lock:
            prev = cls._chain[-1].get("hash","") if cls._chain else ""
            entry = {"id": uid("e-"), "ts": now_iso(), "op": op, "info": info, "prev": prev,
"version": __version__}
            try:
                entry_str = json.dumps(entry, sort_keys=True, ensure_ascii=False)
                entry['hash'] = hashlib.sha256(entry_str.encode('utf-8')).hexdigest()
            except Exception:
                entry['hash'] = uid("h-")
            cls._chain.append(entry)
            try:
                tmp = LEDGER_PATH + ".tmp"
                with open(tmp, "w", encoding="utf-8") as f:
                    json.dump(cls._chain, f, ensure_ascii=False, indent=2)
                os.replace(tmp, LEDGER_PATH)
            except Exception:
                logger.exception("Ledger write failed")
            logger.info("AUDIT %s %s", op, json.dumps(info, default=str))
            return entry['id']

    @classmethod
    def export(cls) -> List[Dict[str,Any]]:
        return list(cls._chain)

# -------------
# Data models
# -------------
@dataclass
class SensorReading:
    id: str
    pollutant: str
    value: float
    unit: str
    ts: float
    location: Tuple[float,float]
```

```
        quality: float = 1.0
        meta: Dict[str,Any] = field(default_factory=dict)
        def to_dict(self) -> Dict[str,Any]:
                return
{"id":self.id,"pollutant":self.pollutant,"value":self.value,"unit":self.unit,"ts":self.ts,"location":sel
f.location,"quality":self.quality,"meta":self.meta}

@dataclass
class PollutantSample:
        pollutant: str
        concentration: float
        unit: str
        location: Tuple[float,float]
        ts: float
        def to_dict(self) -> Dict[str,Any]:
                return
{"pollutant":self.pollutant,"concentration":self.concentration,"unit":self.unit,"location":self.lo
cation,"ts":self.ts}

@dataclass
class TreatmentPlan:
        id: str
        pollutant: str
        method: str
        parameters: Dict[str,Any]
        estimated_cost_usd: float
        estimated_duration_hours: float
        expected_reduction_pct: float
        safety_notes: List[str]
        def to_dict(self) -> Dict[str,Any]:
                return
{"id":self.id,"pollutant":self.pollutant,"method":self.method,"parameters":self.parameters,"es
timated_cost_usd":self.estimated_cost_usd,"estimated_duration_hours":self.estimated_dur
ation_hours,"expected_reduction_pct":self.expected_reduction_pct,"safety_notes":self.safet
y_notes}

# ------------------------
# Vector dims (single source of truth)
# ------------------------
VECTOR_DIM = 256
MATERIAL_EMBED_DIM = 128
INDEX_DIM = VECTOR_DIM + MATERIAL_EMBED_DIM

# ------------------------
# Dependencies
# ——————————-
try:
    import numpy as np
except Exception:
    raise RuntimeError("numpy is required")

# ——————————-
# Nuclide library (10 nuclides)
# ——————————-
NUCLIDE_TABLE = {
    "Cs-137":
{"halflife_years":30.17,"gamma_keV":[661.7],"gamma_constant_Sv_h_per_Bq_at_1m":1.3e-1
7},
    "Sr-90":
{"halflife_years":28.79,"gamma_keV":[],"gamma_constant_Sv_h_per_Bq_at_1m":0.0},
    "Pu-239":
{"halflife_years":24110.0,"gamma_keV":[129.3,375.0],"gamma_constant_Sv_h_per_Bq_at_1
m":5e-18},
    "I-131":
{"halflife_years":0.0238,"gamma_keV":[364.5],"gamma_constant_Sv_h_per_Bq_at_1m":2.2e-
16},
    "Co-60":
{"halflife_years":5.27,"gamma_keV":[1173.2,1332.5],"gamma_constant_Sv_h_per_Bq_at_1m
":3.0e-16},
    "Am-241":
{"halflife_years":432.2,"gamma_keV":[59.5],"gamma_constant_Sv_h_per_Bq_at_1m":9e-18},
    "U-235":
{"halflife_years":7.04e8,"gamma_keV":[185.7],"gamma_constant_Sv_h_per_Bq_at_1m":1e-1
8},
    "U-238":
{"halflife_years":4.468e9,"gamma_keV":[1001.0],"gamma_constant_Sv_h_per_Bq_at_1m":8e
-19},
    "Th-232":
{"halflife_years":1.405e10,"gamma_keV":[238.6],"gamma_constant_Sv_h_per_Bq_at_1m":5e
-19},
    "K-40":
{"halflife_years":1.248e9,"gamma_keV":[1460.8],"gamma_constant_Sv_h_per_Bq_at_1m":4e
-17}
}

# ——————————-
# Utility: normalize and fix vector dimension
# ——————————-
```

```
def normalize_and_fix_dim(vec: "np.ndarray", target_dim: int) ->"np.ndarray":
    v = np.asarray(vec, dtype=float).reshape(-1)
    orig_len = v.size
    if orig_len > target_dim:
        v = v[:target_dim]
    elif orig_len < target_dim:
        pad = np.zeros(target_dim - orig_len, dtype=float)
        v = np.concatenate([v, pad], axis=0)
    norm = np.linalg.norm(v) + 1e-12
    return (v / norm).astype(float)

# -------------
# VectorIndex (defensive)
# -------------
class VectorIndex:
    def __init__(self, dim: int = INDEX_DIM):
        self.dim = dim
        self.lock = threading.RLock()
        self.idmap: Dict[int,str] = {}
        self.revidmap: Dict[str,int] = {}
        self.idtometa: Dict[int,Dict[str,Any]] = {}
        self.next_idx = 0
        self.vectors: List[np.ndarray] = []

    def add(self, uid_str: str, vec: np.ndarray, meta: Dict[str,Any]):
        with self.lock:
            try:
                orig_len = int(np.asarray(vec).reshape(-1).size)
                fixed = normalize_and_fix_dim(vec, self.dim)
                idx = self.next_idx; self.next_idx += 1
                self.idmap[idx] = uid_str; self.revidmap[uid_str] = idx
                self.idtometa[idx] = {"meta": meta, "ts": meta.get("ts", time.time())}
                self.vectors.append(np.array(fixed, dtype=float))
                Ledger.record("INDEXADD", {"uid": uid_str, "idx": idx, "orig_len": orig_len,
"fixed_len": int(fixed.size), "meta": meta})
            except Exception:
                logger.exception("VectorIndex.add failed")
                raise

    def query(self, vec: np.ndarray, topk: int = 10) -> List[Dict[str,Any]]:
        with self.lock:
            try:
                q = normalize_and_fix_dim(vec, self.dim)
                if not self.vectors:
```

```
                return []
            mats = np.stack(self.vectors, axis=0)
            sims = (mats @ q.reshape(-1,1)).reshape(-1)
            idxs = np.argsort(-sims)[:topk]
            res = []
            for i in idxs:
                res.append({"id": self.idmap.get(int(i)), "score": float(sims[i]), "meta":
self.idtometa.get(i)})
            return res
        except Exception:
            logger.exception("VectorIndex.query failed")
            return []

# -------------
# Vectorizers&MaterialDecomposer (simulation)
# -------------
class Vectorizer:
    def __init__(self, dim: int = VECTOR_DIM):
        self.dim = dim
    def transform(self, sample: Dict[str,Any], window: Optional[List[Dict[str,Any]]] = None)
-> np.ndarray:
        phys = sample.get("phys", {})
        vals = [float(phys.get("value", 0.0)), float(phys.get("temp", 0.0)),
float(phys.get("chlf", 0.0))]
        v = np.array(vals + [0.0]*(self.dim - len(vals)), dtype=float)[:self.dim]
        n = np.linalg.norm(v) + 1e-12
        return (v / n).astype(float)

class MaterialVectorizer:
    def __init__(self, embeddim: int = MATERIAL_EMBED_DIM):
        self.embeddim = embeddim
    def transform(self, sample: Dict[str,Any]) -> np.ndarray:
        mat = sample.get("material", {})
        vec = np.array(mat.get("mass_spec", [0.0]*self.embeddim)[:self.embeddim],
dtype=float)
        n = np.linalg.norm(vec) + 1e-12
        return (vec / n).astype(float)

class MaterialDecomposer:
    def __init__(self, nbases: int = 16):
        self.nbases = nbases
        # Simulation basis; production must train on real spectra and persist model
        self.basis = np.abs(np.random.randn(self.nbases, MATERIAL_EMBED_DIM))
    def decompose(self, material_vec: np.ndarray) -> List[float]:
        coeffs = np.maximum(0.0, np.dot(self.basis, material_vec))
        s = np.sum(coeffs) + 1e-12
        return (coeffs / s).tolist()

# ————————————-
# Forensic modules
# ————————————-
class AFMProcessor:
    def __init__(self, smooth_sigma: float = 1.0):
        self.smooth_sigma = float(smooth_sigma)
    def extract_features(self, distance: List[float], force: List[float]) -> Dict[str,Any]:
        try:
            if not force:
                return {"ok": False, "error": "empty", "features": {}}
            arr = np.asarray(force, dtype=float)
            mean_force = float(arr.mean()); max_force = float(arr.max()); std_force =
float(arr.std())
            auc = float(np.trapz(arr)) if arr.size>0 else 0.0
            return {"ok": True, "features": {"mean_force": mean_force, "max_force":
max_force, "std_force": std_force, "auc": auc}}
        except Exception:
            return {"ok": False, "error": "exception", "features": {}}

class ImageProcessor:
    def image_proxy_features(self, arr) -> Dict[str,Any]:
        if arr is None:
            return {"mean": 0.0, "std": 0.0, "hist": []}
        a = np.asarray(arr, dtype=float)
        mean = float(a.mean()); std = float(a.std())
        return {"mean": mean, "std": std, "hist": []}

class SpectrumMatcher:
    def match(self, energies: List[float], nuclide_table: Dict[str,Any], top_k: int = 10) ->
List[Dict[str,Any]]:
        if not energies:
            return []
        candidates=[]
        for name,meta in nuclide_table.items():
            lines = meta.get("gamma_keV", [])
            score = 0.0
            for line in lines:
                matches = sum(1 for e in energies if abs(e - line) <= 5.0)
                score += matches
            score = float(score) / (1.0 + len(energies))
```

```
                score = min(1.0, score + random.uniform(0.0, 0.25))
                candidates.append({"nuclide":name, "score":score})
            candidates = sorted(candidates, key=lambda x: x["score"], reverse=True)
            return [c for c in candidates if c["score"]>0.01][:top_k]

class ForensicFusionEngine:
    def __init__(self, weights: Optional[Dict[str,float]] = None):
        self.weights = weights or {"afm":0.25,"image":0.15,"material":0.4,"spectrum":0.2}
    def fuse(self, evidence: Dict[str,Any]) -> Dict[str,Any]:
        total_w = 0.0; acc = 0.0; breakdown = {}
        for k,w in self.weights.items():
            v = float(evidence.get(k, 0.0)); v = max(0.0, min(1.0, v))
            breakdown[k] = {"score": v, "weight": float(w)}
            acc += v * float(w); total_w += float(w)
        forensic_score = float(acc / total_w) if total_w>0 else 0.0
        Ledger.record("FORENSIC_SCORE",          {"forensic_score":          forensic_score,
"breakdown": breakdown})
        return {"forensic_score": forensic_score, "breakdown": breakdown}

class AFMAgingScorer:
    def __init__(self, config: Optional[Dict[str,float]] = None):
        self.config                    =                    config                    or
{"adhesion_w":0.35,"hysteresis_w":0.25,"slope_w":0.15,"std_w":0.10,"auc_w":0.15}
    def score(self, afm_features: Dict[str,Any]) -> float:
        if not afm_features: return 0.0
        adhesion = afm_features.get("adhesion", afm_features.get("max_force",0.0))
        hysteresis = afm_features.get("hysteresis", 0.0)
        slope = abs(afm_features.get("slope", 0.0))
        std = afm_features.get("std_force", 0.0)
        auc = afm_features.get("auc", 0.0)
        def norm(x, scale=1.0): return max(0.0, min(1.0, float(x)/(scale + 1e-12)))
        s = (self.config["adhesion_w"] * norm(adhesion, scale=10.0) +
             self.config["hysteresis_w"] * norm(hysteresis, scale=1.0) +
             self.config["slope_w"] * norm(slope, scale=1.0) +
             self.config["std_w"] * norm(std, scale=5.0) +
             self.config["auc_w"] * norm(auc, scale=100.0))
        return float(max(0.0, min(1.0, s)))

# -------------
# Digital Twin (expanded)
# -------------
class DigitalTwin:
    """
    Expanded digital twin for atmospheric dispersion and deposition.
    Simplified physics for simulation and decision support only.
    """
    def __init__(self, env: Optional[Dict[str,Any]] = None):
        self.env = env or {"wind_speed": 3.0, "wind_dir_deg": 90.0, "stability_class": "D",
"precipitation_mm_h": 0.0, "terrain_factor": 1.0}
    def step_dispersion(self, sources: List[Dict[str,Any]], grid: List[Tuple[float,float]],
dt_seconds: float):
        metric_inc("sim_steps")
        ws = self.env.get("wind_speed", 3.0)
        wd = math.radians(self.env.get("wind_dir_deg", 90.0))
        wx, wy = ws * math.cos(wd), ws * math.sin(wd)
        stability = self.env.get("stability_class", "D")
        stability_factor = {"A":1.5,"B":1.2,"C":1.0,"D":0.8,"E":0.6,"F":0.4}.get(stability, 0.8)
        precip = self.env.get("precipitation_mm_h", 0.0)
        terrain = self.env.get("terrain_factor", 1.0)
        samples = []
        for loc in grid:
            conc = 0.0
            for src in sources:
                dx = loc[0] - src["location"][0]; dy = loc[1] - src["location"][1]
                r = math.hypot(dx, dy) + 1.0
                adv = 1.0 + max(0.0, (dx*wx + dy*wy) / (r + 1e-6)) * 0.5
                dispersion = src.get("strength", 0.0) * adv / (r**1.6) * stability_factor /
terrain
                conc += dispersion * (dt_seconds / 3600.0)
            if precip > 0:
                washout = 1.0 - min(0.95, 0.05 * precip)
                conc *= washout
            conc *= (1.0 + random.uniform(-0.02, 0.02))
            samples.append({"location": loc, "concentration": max(0.0, conc)})
        return samples

    def simulate_deposition(self, samples: List[Dict[str,Any]], deposition_rate: float = 0.01):
        deposits = []
        for s in samples:
            deposits.append({"location": s["location"], "deposited": s["concentration"] *
deposition_rate})
        return deposits

# ————————————-
# Source estimator (map-inverse)
# ————————————-
class SourceEstimator:
    def _G_rad(self, sensors, grid):
        m = len(sensors); n = len(grid); G = [[0.0]*n for _ in range(m)]
        for i,(sloc,_) in enumerate(sensors):
            for j,gloc in enumerate(grid):
                r = math.hypot(sloc[0]-gloc[0], sloc[1]-gloc[1]) + 0.1
                G[i][j] = 1.0/(r**2.0)
        return G
    def map_inverse(self, samples: List[PollutantSample], grid: List[Tuple[float,float]], reg_lambda: float = 1e-3):
        metric_inc("source_est_calls")
        if not samples or not grid: return {"candidates": [], "model": "map_v1"}
        sensors = [(s.location, s.concentration) for s in samples]
        G = self._G_rad(sensors, grid)
        d = [s[1] for s in sensors]; n = len(grid)
        GtG = [[0.0]*n for _ in range(n)]; Gtd = [0.0]*n
        for j in range(n):
            for k in range(n):
                ssum = 0.0
                for i in range(len(sensors)): ssum += G[i][j]*G[i][k]
                GtG[j][k] = ssum
            s2 = 0.0
            for i in range(len(sensors)): s2 += G[i][j]*d[i]
            Gtd[j] = s2
        for j in range(n): GtG[j][j] += reg_lambda
        try:
            q = self._solve_linear(GtG, Gtd)
        except Exception:
            q = [0.0]*n
        candidates = []
        for j,gloc in enumerate(grid):
            candidates.append({"location": gloc, "strength": float(max(0.0, q[j])), "confidence": 0.5, "model": "map_v1"})
        return {"candidates": candidates, "model": "map_v1"}
    def _solve_linear(self, A, b):
        n = len(b); M = [row[:] for row in A]; rhs = b[:]
        for k in range(n):
            piv = k
            for i in range(k,n):
                if abs(M[i][k]) > abs(M[piv][k]): piv = i
            if abs(M[piv][k]) < 1e-12: continue
            M[k], M[piv] = M[piv], M[k]; rhs[k], rhs[piv] = rhs[piv], rhs[k]
            fac = M[k][k]; M[k] = [x/fac for x in M[k]]; rhs[k] /= fac
            for i in range(n):
                if i == k: continue
                fac2 = M[i][k]
                if abs(fac2) < 1e-15: continue
                M[i] = [M[i][j] - fac2*M[k][j] for j in range(n)]
                rhs[i] -= fac2*rhs[k]
        return rhs

# ——————————
# Dose&risk (relative scoring)
# ——————————
class DecayDoseCalculator:
    def __init__(self, nuclide_table: Dict[str,Any]): self.nuclides = nuclide_table
    def dose_rate_point_Sv_h(self, name: str, activity_bq: float, distance_m: float):
        if distance_m <= 0: distance_m = 0.1
        nu = self.nuclides.get(name, {})
        gamma_const = float(nu.get("gamma_constant_Sv_h_per_Bq_at_1m", 0.0))
        dose = gamma_const * activity_bq / (distance_m**2)
        return float(dose)

class RiskAssessor:
    def __init__(self, pollutant_lib: Dict[str,Any], nuclide_table: Dict[str,Any]):
        self.lib = pollutant_lib; self.decay = DecayDoseCalculator(nuclide_table)
    def score_multi(self, per_nuclide_estimates: Dict[str,Any], exposure_hours: float = 1.0):
        metric_inc("risk_checks")
        loc_map = {}
        for nuclide, est in per_nuclide_estimates.items():
            for c in est.get("candidates", []):
                loc = tuple(c["location"])
                loc_map.setdefault(loc, {})[nuclide] = c.get("strength", 0.0)
        locations = []; total_risk = 0.0
        for loc, nu_map in loc_map.items():
            dose_sum = 0.0; by_nuclide = {}
            for nu, strength in nu_map.items():
                dose = self.decay.dose_rate_point_Sv_h(nu, strength, distance_m=1.0)
                by_nuclide[nu] = dose; dose_sum += dose
            risk_metric = dose_sum * 1e3
            total_risk += risk_metric
            locations.append({"location": loc, "dose_Sv_h": dose_sum, "by_nuclide":
by_nuclide, "risk_metric": risk_metric})
        score = 10.0 if total_risk <= 0 else max(0.0, 10.0 - math.log1p(total_risk)/2.0)
        return {"locations": locations, "total_risk": total_risk, "safety_score": score}

# ——————————
# Strategy generator (suggestions only)
# ——————————
class StrategyGenerator:
    def __init__(self, treatment_lib: Dict[str,Any], pollutant_lib: Dict[str,Any], nuclide_table:
Dict[str,Any]):
        self.tlib = treatment_lib; self.plib = pollutant_lib; self.nuclide = nuclide_table
    def propose(self, pollutant: str, volume_m3: float, concentration: float, robot_profile:
Optional[Dict[str,Any]] = None):
        metric_inc("strategy_calls")
        methods = self.plib.get(pollutant, {}).get("methods",
["containment","isolation","removal","immobilization","phytoremediation"])
        candidates = []
        for m in methods:
            meta = self.tlib.get(m, {"cost_index":100, "efficiency_pct":50})
            eff = float(meta.get("efficiency_pct", 50)) / 100.0
            cost = float(meta.get("cost_index", 100.0)) * float(volume_m3)
            duration = max(1.0, float(volume_m3) * 0.5)
            notes = ["simulation suggestion only", "requires human authorization", "waste
disposal plan required"]
            robot_dose = 0.0
            if robot_profile:
                ambient = robot_profile.get("ambient_dose_Sv_h", 0.0); shield =
robot_profile.get("shielding_factor", 1.0)
                robot_dose = ambient * duration * shield
                cost += robot_profile.get("robot_operational_cost_usd_per_hour", 100.0)
* duration
            plan = TreatmentPlan(id=str(uuid.uuid4()), pollutant=pollutant, method=m,
parameters={"volume_m3":float(volume_m3)}, estimated_cost_usd=float(cost),
estimated_duration_hours=float(duration), expected_reduction_pct=float(eff*100.0),
safety_notes=notes)
            candidates.append({"plan": plan, "numeric": {"cost_usd": cost, "duration_h":
duration, "efficiency_pct": eff*100.0, "robot_dose_Sv": robot_dose}})
        def compute_score(numeric):
            cost_norm = 1.0 / (1.0 + math.log1p(max(1.0, numeric["cost_usd"])))
            eff_norm = numeric["efficiency_pct"] / 100.0
            robot_penalty = 1.0 / (1.0 + numeric["robot_dose_Sv"]*1e3)
            return (0.6*eff_norm) + (0.3*cost_norm) + (0.1*robot_penalty)
        for c in candidates:
            c["numeric"]["score"] = compute_score(c["numeric"])
        candidates.sort(key=lambda x: x["numeric"]["score"], reverse=True)
        return candidates

# ---------------
# FusionEngine (fixed: added definition)
# ---------------
class FusionEngine:
    """
    Simple fusion engine for sensor readings.
    In production, replace with robust spatio-temporal fusion and sensor models.
    """
    def fuse_readings(self, readings: List[SensorReading]) -> List[PollutantSample]:
        metric_inc("fusion_calls")
        fused = {}
        for r in readings:
            key = (r.pollutant, r.location)
            fused[key] = fused.get(key, 0.0) + r.value * r.quality
        out=[]
        for (pollutant, loc), val in fused.items():
            out.append(PollutantSample(pollutant=pollutant, concentration=float(val),
unit="Bq/m3", location=loc, ts=time.time()))
        return out

# ——————-
# EnhancedScanner (integrates vectorizers, forensic modules)
# ——————-
class EnhancedScanner:
    def __init__(self, engine):
        self.engine = engine
        self.vectorizer = Vectorizer(dim=VECTOR_DIM)
        self.material_vectorizer =
MaterialVectorizer(embeddim=MATERIAL_EMBED_DIM)
        self.decomposer = MaterialDecomposer(nbases=16)
        self.index = VectorIndex(dim=INDEX_DIM)
        self.afm = AFMProcessor()
        self.img = ImageProcessor()
        self.spectrum = SpectrumMatcher()
        self.forensic = ForensicFusionEngine()
        self.afm_aging = AFMAgingScorer()
        self.lock = threading.RLock()

    def ingest_and_index(self, sample: Dict[str,Any], window: Optional[List[Dict[str,Any]]] =
None) -> bool:
        try:
            v = self.vectorizer.transform(sample, window=window)
            m = self.material_vectorizer.transform(sample)
            try:
                combined_raw = np.concatenate([v, m], axis=0)
            except Exception:
                combined_raw = v
            fixed = normalize_and_fix_dim(combined_raw, self.index.dim)
            uid_str = sample.get("id", uid("s-"))
            meta = {"ts": sample.get("ts", time.time()), "device": sample.get("meta",
{}).get("deviceid")}
            self.index.add(uid_str, fixed, meta)
            Ledger.record("INDEX_INGEST", {"uid": uid_str, "orig_len":
int(np.asarray(combined_raw).reshape(-1).size), "fixed_len": int(fixed.size), "meta": meta})
            return True
        except Exception:
            logger.exception("EnhancedScanner.ingest_and_index failed")
            return False

    def detect_anomalies(self, recent_window: List[Dict[str,Any]], topk: int = 10) ->
List[Dict[str,Any]]:
        metric_inc("sim_steps")
        if not recent_window: return []
        vecs = [self.vectorizer.transform(s, window=recent_window) for s in
recent_window]
        cur_phys = np.mean(np.stack(vecs, axis=0), axis=0)
        mat_vecs = [self.material_vectorizer.transform(s) for s in recent_window]
        avg_mat = np.mean(np.stack(mat_vecs, axis=0), axis=0) if mat_vecs else
np.zeros(MATERIAL_EMBED_DIM, dtype=float)
        combined_query = np.concatenate([cur_phys, avg_mat], axis=0)
        sims = self.index.query(combined_query, topk=topk)
        avg_score = np.mean([r["score"] for r in sims]) if sims else 0.0
        anomaly_score = max(0.0, 1.0 - avg_score)
        afm_scores=[]; image_scores=[]; material_conf=[]; spectrum_scores=[]
        for s in recent_window:
            afm_meta = s.get("meta", {}).get("afm")
            if afm_meta:
                afm_res = self.afm.extract_features(afm_meta.get("distance", []),
afm_meta.get("force", []))
                if afm_res.get("ok"):
                    afm_scores.append(self.afm_aging.score(afm_res["features"]))
            img_arr = s.get("meta", {}).get("image_array")
            if img_arr is not None:
                img_feat = self.img.image_proxy_features(img_arr)
                image_scores.append(min(1.0, img_feat.get("mean", 0.0)))
            mvec = self.material_vectorizer.transform(s)
            coeffs = self.decomposer.decompose(mvec)
            material_conf.append(max(coeffs) if coeffs else 0.0)
            energies = s.get("meta", {}).get("spectrum_energies", [])
            spec_cands = self.spectrum.match(energies, NUCLIDE_TABLE)
            spectrum_scores.append(spec_cands[0]["score"] if spec_cands else 0.0)
        evidence = {"afm": float(np.mean(afm_scores)) if afm_scores else 0.0, "image":
float(np.mean(image_scores)) if image_scores else 0.0, "material":
```

```
float(np.mean(material_conf)) if material_conf else 0.0, "spectrum":
float(np.mean(spectrum_scores)) if spectrum_scores else 0.0}
        forensic = self.forensic.fuse(evidence)
        lats = [s.get("meta", {}).get("gps", {}).get("lat", 0.0) for s in recent_window]
        lons = [s.get("meta", {}).get("gps", {}).get("lon", 0.0) for s in recent_window]
        avg_loc = (float(sum(lats)/len(lats)), float(sum(lons)/len(lons))) if lats and lons
else (0.0,0.0)
        candidate = {"id": uid("cand-"), "location": avg_loc, "anomaly_score":
float(anomaly_score), "forensic_score": forensic["forensic_score"], "evidence": evidence,
"similar_history": sims, "evidence_count": len(recent_window), "timestamp": now_iso()}
        Ledger.record("DARKCANDIDATE", {"id": candidate["id"], "loc":
candidate["location"], "anomaly_score": candidate["anomaly_score"], "forensic_score":
candidate["forensic_score"]})
        return [candidate]

    def refine_candidates_with_materials(self, per_nuclide_est: Dict[str,Any],
dark_candidates: List[Dict[str,Any]]) -> Dict[str,Any]:
        enhanced = {}
        for cand in dark_candidates:
            loc = tuple(cand["location"])
            merged = {"location": loc, "evidence": cand["evidence_count"], "material_sig":
[], "nuclide_scores": {}}
            for nu, est in per_nuclide_est.items():
                best=None; best_dist=float("inf")
                for c in est.get("candidates", []):
                    d = math.hypot(c["location"][0]-loc[0], c["location"][1]-loc[1])
                    if d < best_dist:
                        best_dist = d; best = c
                if best:
                    dist_factor = math.exp(-best_dist/100.0)
                    fused_conf = best.get("confidence", 0.5) * dist_factor
                    merged["nuclide_scores"][nu] = {"strength": best.get("strength",0.0),
"fused_conf": float(fused_conf), "dist_m": float(best_dist)}
            max_conf = max([v["fused_conf"] for v in merged["nuclide_scores"].values()])
if merged["nuclide_scores"] else 0.0
            priority = 0.5 * cand.get("anomaly_score",0.0) + 0.35 *
cand.get("forensic_score",0.0) + 0.15 * max_conf
            merged["priority_score"] = float(priority)
            enhanced[cand["id"]] = merged
            Ledger.record("CANDIDATE_REFINED", {"id": cand["id"], "priority":
merged["priority_score"], "nuclides": list(merged["nuclide_scores"].keys())})
        return enhanced

    def adaptive_sampling_plan(self, enhanced_candidates: Dict[str,Any],
available_devices: List[Dict[str,Any]], max_assign: int = 5) -> List[Dict[str,Any]]:
        plans=[]
        sorted_cands = sorted(enhanced_candidates.items(), key=lambda x: x[1]["priority_score"], reverse=True)
        assigned=0
        for cid, info in sorted_cands:
            if assigned >= max_assign: break
            best_dev=None; best_score=-1.0
            for d in available_devices:
                dev_pos = d.get("pos",(0.0,0.0))
                dist = math.hypot(dev_pos[0]-info["location"][0], dev_pos[1]-info["location"][1])
                battery = d.get("battery_pct",50.0)
                if battery < 20.0: continue
                score = (1.0/(1.0+dist)) * (battery/100.0)
                if score > best_score:
                    best_score = score; best_dev = d
            plan = {"candidate_id": cid, "target_location": info["location"], "suggested_device": best_dev.get("device_id") if best_dev else None, "priority": info["priority_score"], "safety_notes": ["human authorization required","do not execute without certified driver","waste disposal plan required"], "timestamp": now_iso()}
            Ledger.record("ADAPTIVE_PLAN_CREATED", {"plan_id": uid("plan-"), "candidate": cid, "suggested_device": plan["suggested_device"], "priority": plan["priority"]})
            plans.append(plan)
            assigned += 1
        return plans

# -------------
# Device templates and secure execution placeholders
# -------------
class DeviceAdapterBase:
    def telemetry(self) -> Dict[str,Any]:
        raise NotImplementedError()
    def health(self) -> Dict[str,Any]:
        raise NotImplementedError()
    def propose_task(self, task: Dict[str,Any]) -> Dict[str,Any]:
        raise NotImplementedError()
    def execute_task(self, task: Dict[str,Any], auth_token: Optional[str] = None) -> Dict[str,Any]:
        raise NotImplementedError("execute_task disabled in safe prototype; implement in certified driver with HSM-backed authorization")

class HeavyDutyDrone(DeviceAdapterBase):
    def __init__(self, device_id: str, capabilities: Dict[str,Any]):
        self.device_id = device_id; self.capabilities = capabilities
    def telemetry(self):
        return {"device_id": self.device_id, "type": "heavy_drone", "pos":
(random.uniform(-1000,1000), random.uniform(-1000,1000)), "battery_pct":
random.uniform(20,100), "capabilities": self.capabilities}
    def health(self):
        return {"device_id": self.device_id, "status": "ok", "cumulative_dose_Sv": 0.0,
"instant_dose_Sv_h": 0.0}
    def propose_task(self, task: Dict[str,Any]) -> Dict[str,Any]:
        return {"device_id": self.device_id, "feasible": True, "reasons": []}
    def execute_task(self, task: Dict[str,Any], auth_token: Optional[str] = None) ->
Dict[str,Any]:
        raise NotImplementedError("execute_task disabled in safe prototype")

class HeavyDutyGroundRobot(DeviceAdapterBase):
    def __init__(self, device_id: str, capabilities: Dict[str,Any]):
        self.device_id = device_id; self.capabilities = capabilities
    def telemetry(self):
        return {"device_id": self.device_id, "type": "heavy_ground_robot", "pos":
(random.uniform(-500,500), random.uniform(-500,500)), "battery_pct":
random.uniform(20,100), "capabilities": self.capabilities}
    def health(self):
        return {"device_id": self.device_id, "status": "ok", "cumulative_dose_Sv": 0.0,
"instant_dose_Sv_h": 0.0}
    def propose_task(self, task: Dict[str,Any]) -> Dict[str,Any]:
        return {"device_id": self.device_id, "feasible": True, "reasons": []}
    def execute_task(self, task: Dict[str,Any], auth_token: Optional[str] = None) ->
Dict[str,Any]:
        raise NotImplementedError("execute_task disabled in safe prototype")

# ————————————-
# HSM / Authorization placeholders (no keys, simulation only)
# ————————————-
class AuthorizationManager:
    """
    Placeholder for three-layer safety valve:
      1) Policy review (automated checks)
      2) Dual human authorization (two approvers)
      3) HSM signature (placeholder)
    Real implementation must integrate with enterprise HSM/KMS and identity provider.
    """
    def __init__(self):
        self.pending: Dict[str, Dict[str,Any]] = {}
    def policy_check(self, task_package: Dict[str,Any]) -> Tuple[bool,str]:
        if task_package.get("requires_authorization", True) is False:
            return False, "task must require authorization"
        if task_package.get("plan", {}).get("parameters", {}).get("unsafe", False):
            return False, "unsafe parameter flagged"
        return True, "policy_ok"
    def request_approval(self, task_package: Dict[str,Any]) -> str:
        pid = uid("auth-")
        self.pending[pid] = {"task": task_package, "approvals": [], "created": now_iso()}
        Ledger.record("AUTH_REQUESTED", {"auth_id": pid, "task_id":
task_package.get("task_id")})
        return pid
    def approve(self, auth_id: str, approver_id: str) -> Dict[str,Any]:
        rec = self.pending.get(auth_id)
        if not rec:
            return {"ok": False, "reason": "not_found"}
        if approver_id in rec["approvals"]:
            return {"ok": False, "reason": "already_approved"}
        rec["approvals"].append(approver_id)
        Ledger.record("AUTH_APPROVAL", {"auth_id": auth_id, "approver": approver_id})
        if len(rec["approvals"]) >= 2:
            rec["signed"] = True
            rec["signature"] = "HSM-SIGNATURE-PLACEHOLDER-" +
sha256_of(rec["task"])
            Ledger.record("AUTH_SIGNED", {"auth_id": auth_id, "signature":
rec["signature"]})
        return {"ok": True, "approvals": rec["approvals"], "signed": rec.get("signed", False)}
    def get_status(self, auth_id: str) -> Dict[str,Any]:
        rec = self.pending.get(auth_id)
        if not rec:
            return {"ok": False, "reason": "not_found"}
        return {"ok": True, "approvals": rec.get("approvals", []), "signed": rec.get("signed",
False), "signature": rec.get("signature")}

auth_manager = AuthorizationManager()

# ----------------
# PurifyEngine (orchestration)
# ----------------
class PurifyEngine:
    def __init__(self, config: Optional[Dict[str,Any]] = None):
        self.config =
{"default_volume_m3":100.0,"exposure_hours":1.0,"control_enabled":False,"mc_runs":200,"
max_candidate_nuclides":20}
        if config: self.config.update(config)
        self.nuclide_table = NUCLIDE_TABLE
        self.pollutant_lib = {k:
{"unit":"Bq/m3","methods":["containment","isolation","removal","immobilization","phytoreme
diation"]} for k in self.nuclide_table.keys()}
        self.treatment_lib =
{"containment":{"cost_index":1.0,"efficiency_pct":90},"isolation":{"cost_index":1.5,"efficiency
_pct":95},"removal":{"cost_index":3.0,"efficiency_pct":80},"immobilization":{"cost_index":2.0,
"efficiency_pct":70},"phytoremediation":{"cost_index":0.8,"efficiency_pct":40}}
        self.devices: List[Any] = []
        self.fusion = FusionEngine()
        self.estimator = SourceEstimator()
        self.risk = RiskAssessor(self.pollutant_lib, self.nuclide_table)
        self.strategy = StrategyGenerator(self.treatment_lib, self.pollutant_lib,
self.nuclide_table)
        self.sim = DigitalTwin()
        self.enhanced = EnhancedScanner(self)
        Ledger.record("engine_init", {"version": __version__, "vector_dim": VECTOR_DIM,
"material_emb_dim": MATERIAL_EMBED_DIM, "index_dim": INDEX_DIM})

    def register_device(self, dev: Any):
        self.devices.append(dev)
        Ledger.record("device_registered", {"device_type": dev.__class__.__name__,
"device_id": getattr(dev, "device_id", str(dev))})

    def ingest(self, readings: Optional[List[Dict[str,Any]]] = None) -> List[SensorReading]:
        if readings is not None:
            out=[]
            for r in readings:
                try:
                    sr = SensorReading(id=r.get("id", uid("r-")),
pollutant=r.get("pollutant","unknown"), value=float(r.get("value",0.0)),
unit=r.get("unit","Bq/m3"), ts=float(r.get("ts", time.time())),
location=tuple(r.get("location",(0.0,0.0))), quality=float(r.get("quality",1.0)),
meta=r.get("meta",{}))
                    out.append(sr)
                except Exception:
                    logger.exception("invalid reading")
            return out
        out=[]
        for dev in self.devices:
            try:
                tel = dev.telemetry()
                loc = tel.get("pos",(0.0,0.0))
                pollutant = random.choice(list(self.nuclide_table.keys()))
                val = random.uniform(0.1,10.0)
                sr = SensorReading(id=tel.get("device_id", uid("dev-")),
pollutant=pollutant, value=val, unit="Bq/m3", ts=time.time(), location=loc, quality=0.9,
meta={"spectrum_energies": None})
                out.append(sr)
            except Exception:
                logger.exception("device telemetry failed")
        return out

    def fuse(self, readings: List[SensorReading]) -> List[PollutantSample]:
        return self.fusion.fuse_readings(readings)

    def multi_nuclide_candidates(self, readings: List[SensorReading], top_k: int = 20) ->
List[str]:
        score_map={}
        for r in readings:
            score_map[r.pollutant] = max(score_map.get(r.pollutant,0.0), 0.5)
        if not score_map:
            return list(self.nuclide_table.keys())[:top_k]
        sorted_nuclides = sorted(score_map.items(), key=lambda x: x[1], reverse=True)
        return [n for n,_ in sorted_nuclides][:top_k]

    def estimate_multi(self, fused_samples: List[PollutantSample], candidate_nuclides:
List[str]) -> Dict[str,Any]:
        per_nuclide={}
        def worker(nuclide):
            samples = [PollutantSample(pollutant=nuclide,
concentration=s.concentration, unit=s.unit, location=s.location, ts=s.ts) for s in
fused_samples]
            grid=[]; seen=set()
            for s in samples:
                x0,y0 = s.location
                for i in range(-3,4):
                    for j in range(-3,4):
                        pt=(round(x0 + i*50.0,6), round(y0 + j*50.0,6))
                        if pt not in seen:
                            seen.add(pt); grid.append(pt)
            est = self.estimator.map_inverse(samples, grid, reg_lambda=1e-4)
            return nuclide, est
        max_workers = min(8, max(1, len(candidate_nuclides)))
        with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as ex:
            futures = [ex.submit(worker, n) for n in candidate_nuclides]
            for f in concurrent.futures.as_completed(futures):
                try:
                    nu, est = f.result(); per_nuclide[nu] = est
                except Exception:
                    logger.exception("estimate worker failed")
        return per_nuclide

    def monte_carlo_multi(self, fused_samples: List[PollutantSample],
per_nuclide_grid_map: Dict[str,List[Tuple[float,float]]], mc_runs: int = 200) -> Dict[str,Any]:
        metric_inc("mc_runs", mc_runs)
        per_nuclide_summary={}
        def mc_worker(nuclide, grid):
            results=[]
            for run in range(mc_runs):
                perturbed=[]
                for s in fused_samples:
                    sigma=max(1e-6, 0.1*s.concentration)
                    noise=random.gauss(0, sigma)
                    perturbed.append(PollutantSample(pollutant=nuclide,
concentration=max(0.0, s.concentration+noise), unit=s.unit, location=s.location, ts=s.ts))
                est = self.estimator.map_inverse(perturbed, grid, reg_lambda=1e-3)
                strengths=[c["strength"] for c in est.get("candidates",[])]
                results.append(strengths)
            if not results: return {"mc":[],"summary":[]}
            transposed=list(zip(*results))
            summary=[]
            for arr in transposed:
                arr_list=list(arr)
                med=statistics.median(arr_list)
                p05=min(arr_list) if len(arr_list)<20 else
statistics.quantiles(arr_list,n=20)[0]
                p95=max(arr_list) if len(arr_list)<20 else
statistics.quantiles(arr_list,n=20)[-1]
                summary.append({"median":med,"p05":p05,"p95":p95})
            return {"mc":results,"summary":summary}
        max_workers = min(8, max(1, len(per_nuclide_grid_map)))
        with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as ex:
            futures = {ex.submit(mc_worker, nu, grid): nu for nu,grid in
per_nuclide_grid_map.items()}
            for f in concurrent.futures.as_completed(futures):
                nu = futures[f]
                try:
                    per_nuclide_summary[nu] = f.result()
                except Exception:
                    logger.exception("mc worker failed for %s", nu)
        return per_nuclide_summary
    def make_task_package(self, plan: Dict[str,Any], nuclide: str, safety_info: Dict[str,Any],
model_hash: str) -> Dict[str,Any]:
        task = {"task_id": str(uuid.uuid4()), "nuclide": nuclide, "plan": plan, "safety":
safety_info, "model_hash": model_hash, "timestamp": now_iso(), "requires_authorization":
True}
        Ledger.record("task_package_created", {"task_id": task["task_id"], "nuclide":
nuclide})
        return task

    def run_cycle(self, readings: Optional[List[Dict[str,Any]]] = None, dry_run: bool = True,
run_sensitivity: bool = False) -> Dict[str,Any]:
        run_id = Ledger.record("cycle_start", {"dry_run": dry_run})
        try:
            raw = self.ingest(readings)
            Ledger.record("ingest", {"count": len(raw)})
            if not raw:
                return {"status":"no_data","run_id":run_id}
            candidate_nuclides = self.multi_nuclide_candidates(raw,
top_k=self.config.get("max_candidate_nuclides",20))
            Ledger.record("candidates_from_spectra", {"candidates":
candidate_nuclides})
            fused = self.fuse(raw)
            Ledger.record("fusion", {"count": len(fused)})
            per_nuclide_est = self.estimate_multi(fused, candidate_nuclides)
            Ledger.record("multi_estimate", {"nuclides": list(per_nuclide_est.keys())})
            per_nuclide_grid_map = {nu: [tuple(c["location"]) for c in
est.get("candidates",[])] for nu,est in per_nuclide_est.items()}
            mc_results = self.monte_carlo_multi(fused, per_nuclide_grid_map,
mc_runs=self.config.get("mc_runs",200))
            Ledger.record("mc_complete", {"nuclides": list(mc_results.keys())})
            aggregated = self.risk.score_multi(per_nuclide_est,
exposure_hours=self.config.get("exposure_hours",1.0))
            Ledger.record("aggregated_risk", {"safety_score":
aggregated.get("safety_score"), "total_risk": aggregated.get("total_risk")})
            strategy_tables = {}; chosen_plans = {}
            for nu in per_nuclide_est.keys():
                strengths = [c.get("strength",0.0) for c in
per_nuclide_est[nu].get("candidates",[])]
                avg = statistics.median(strengths) if strengths else 0.0
                candidates = self.strategy.propose(nu,
self.config.get("default_volume_m3",100.0), avg,
robot_profile={"ambient_dose_Sv_h":0.0,"shielding_factor":1.0})
                strategy_tables[nu] =
[{"plan_id":c["plan"].id,"method":c["plan"].method,"score":c["numeric"]["score"]} for c in
candidates]
                chosen_plans[nu] = candidates[0]["plan"].to_dict() if candidates else
None
            Ledger.record("strategies_generated", {"nuclides":
list(strategy_tables.keys())})
            # EnhancedScanner integration
            try:
                recent_window = [r.to_dict() for r in raw][-min(len(raw),12):]
                for s in recent_window:
                    try:
                        self.enhanced.ingest_and_index(s, window=recent_window)
                    except Exception:
                        logger.exception("index ingest failed")
                dark_cands = self.enhanced.detect_anomalies(recent_window, topk=10)
                enhanced =
self.enhanced.refine_candidates_with_materials(per_nuclide_est, dark_cands)
                available_devices = []
                for d in self.devices:
                    try:
                        tel = d.telemetry()
                        available_devices.append({"device_id": tel.get("device_id",
getattr(d,"device_id",str(d))), "type": tel.get("type"), "pos": tel.get("pos"), "battery_pct":
tel.get("battery_pct",50.0)})
                    except Exception:
                        continue
                adaptive_plans = self.enhanced.adaptive_sampling_plan(enhanced,
available_devices, max_assign=5)
                Ledger.record("adaptive_plans_generated", {"count":
len(adaptive_plans)})
            except Exception:
                logger.exception("EnhancedScanner integration failed")
                dark_cands = []; enhanced = {}; adaptive_plans = []
            report = {
                "status":"ok",
                "run_id": run_id,
                "candidate_nuclides": candidate_nuclides,
                "per_nuclide_est": per_nuclide_est,
                "mc_results_summary": {k:v.get("summary") for k,v in
mc_results.items()},
                "aggregated_risk": aggregated,
                "strategy_tables": strategy_tables,
                "chosen_plans": chosen_plans,
                "dark_candidates": dark_cands,
                "enhanced_candidates": enhanced,
                "adaptive_plans": adaptive_plans,
                "audit": Ledger.export(),
                "timestamp": now_iso(),
                "version": __version__
            }
            Ledger.record("cycle_end", {"status":"ok","run_id": run_id})
            return report
        except Exception as e:
            logger.exception("run_cycle error")
            Ledger.record("cycle_error", {"error": str(e), "trace": traceback.format_exc()})
            return
{"status":"error","reason":str(e),"trace":traceback.format_exc(),"run_id":run_id}

# ——————————-
# Edge gateway&HIL placeholders
# ——————————-
class EdgeGateway:
    def __init__(self, broker: str = "mqtt://broker.local"):
        self.broker = broker; self.running = False
    def start(self):
        logger.info("EdgeGateway start placeholder. Broker: %s", self.broker); self.running
= True
    def stop(self):
        logger.info("EdgeGateway stop placeholder."); self.running = False

class HILTestCase:
    def __init__(self, name: str, steps: List[Dict[str,Any]]):
        self.name = name; self.steps = steps
    def run(self, engine: PurifyEngine):
        logger.info("HILTestCase %s start", self.name)
        results=[]
        for step in self.steps:
            if step["action"]=="run_cycle":
                rep = engine.run_cycle(**step.get("params",{}))
                results.append(rep)
            time.sleep(step.get("delay",0.2))
        logger.info("HILTestCase %s complete", self.name)
        return {"name": self.name, "results_summary_keys":[list(r.keys()) for r in results]}

def build_hil_suite():
    tc1                                                                              =
HILTestCase("basic_cycle",[{"action":"run_cycle","params":{"dry_run":True},"delay":0.2}])
    tc2                                                                              =
HILTestCase("forensic_path",[{"action":"run_cycle","params":{"dry_run":True},"delay":0.2}])
    tc3                                                                                    =
HILTestCase("dimension_consistency",[{"action":"run_cycle","params":{"dry_run":True},"dela
y":0.1}])
    return [tc1, tc2, tc3]

# —————————-
# Self-test (safe)
# —————————-
def self_test():
    print("Running purify_enterprise_ultimate_safe_fixed self-test (simulation only)...")
    engine = PurifyEngine()
    engine.register_device(HeavyDutyDrone("drone_1",
{"max_flight_time_min":120,"payload_kg":50}))
    engine.register_device(HeavyDutyGroundRobot("robot_1", {"max_payload_kg":500}))
    edge = EdgeGateway(); edge.start()
    # dimension check
    assert VECTOR_DIM + MATERIAL_EMBED_DIM == INDEX_DIM, "Index dimension
mismatch"
    Ledger.record("selftest_dim_check", {"vector_dim": VECTOR_DIM, "material_emb_dim":
MATERIAL_EMBED_DIM, "index_dim": INDEX_DIM})
    # synthetic sample with AFM and material
    sample = {"id": uid("r-"), "pollutant":"Cs-137", "value": 5.0, "unit":"Bq/m3", "ts":
time.time(), "location": (31.2,121.5), "quality":0.95, "meta": {"deviceid":"dev1",
"gps":{"lat":31.2,"lon":121.5}, "afm": {"distance":[0.0,0.1,0.2], "force":[0.0,0.5,1.0]},
"image_array": None, "material": {"mass_spec":[random.random() for _ in
range(MATERIAL_EMBED_DIM)]}, "spectrum_energies":[661.7]}}
    rep = engine.run_cycle([sample], dry_run=True)
    print("Self-test report keys:", list(rep.keys()))
    print("Ledger entries:", len(Ledger.export()))
    hil = build_hil_suite()
    hil_results = [tc.run(engine) for tc in hil]
    print("HIL results:", hil_results)
    edge.stop()
    print("Metrics:", json.dumps(_METRICS, indent=2))
    print("Self-test complete. Ledger written to", LEDGER_PATH)

# —————————-
# CLI
# —————————-
def _usage():
    print("Usage: python purify_enterprise_ultimate_safe_fixed.py    (no args runs
self-test)")
```

```
if __name__ == "__main__":
    try:
        if len(sys.argv) == 1:
            self_test(); sys.exit(0)
        cmd = sys.argv[1].lower()
        if cmd in ("self-test","selftest","test","demo"):
            self_test()
        else:
            _usage()
    except Exception as e:
        logger.exception("Fatal error: %s", e)
        print("Fatal error occurred. See logs.")
```