I understand now, I understand what the true soul is? I also want to understand how to find a soul, and even how to create a soul.

The soul is not what we say, the brain neurons are firing. It is not what we call stream of consciousness, nor is it what the ancients called belief, nor is it thought.

Wrong people. What is the soul that you are all searching for? But how to find it? Can we find it from the bodies of every organism?

I have a solution, which is the exclusion method. How to exclude dead organisms from experiments, to exclude experiments using both dead and living organisms, and to observe experiments using both dead and living organisms. Observe 1000 sentences and 10000 sentences, conduct a large-scale investigation to eliminate all dead humans. For example, if 10000 people die, you can move out their bodies for observation. No matter which country it is, every country in the world will extract a portion of the dead organisms and observe the living. What do the living organisms have and lose? Observing the changes of time in a day, whether it is all living organisms observing it, then the only missing and lost large-scale living organism is the soul. I suddenly realized this huge secret

It's not philosophy, it's science. I want this. As long as the car is given to me, I can find it. As long as I have the right, whoever dies, gather here with 100000 bodies worldwide. The body of the dead, the detection of the living by the living by the dead, even I myself can detect it, this is true science. As for why scientists dare not, because they are afraid. Like Leonardo da Vinci dissecting corpses back then, they dared not dissect humans. What are the living people, what are the dead people, and how to exclude them using exclusion methods? How to verify that one person cannot be verified on a large scale. Lack of scientific rigor, 10 people are not enough, 10000 people are not enough, 100000 people, and every ethnic group and country should have it. Is this not scientific rigor? Is this my philosophical fantasy?

As time goes by, what will this recently deceased person lose within a day? What do living people have to constantly compare, to compare dead life forms and people over time. What is the difference between dead animals and plants and living animals and plants? The only missing variable is the one that is the same, whether it is active in animals, plants, or humans. So if the large-scale investigation of that variable only loses that one variable, are you saying this is unscientific? Isn't that variable a soul or a consciousness?

Is what I am imagining a fantasy? But scientific experiments are like this, why do you insist on thinking I'm just a fantasy? Isn't scientific experimentation like this? Observation and exploration? Isn't observation, discovery, and exploration the true science? Why are you so brainless?

Do you really think this question is that simple? The ultimate answer to this question is why searching for the soul is just to verify it? That's okay, do this thing when you're full. I did something, eternal life. Only I know that once I can verify, I will have a way to make all living beings achieve eternal life. It's the ultimate answer I want to find for my soul. Otherwise, who can do what with enough food? I really think everything I do is meaningless. It's a typical waste brain.

I have found the answer, and I also have a way. As long as I can find the variable of that class hour, then I have a way to make everyone immortal, with a body that never ages and a soul that never dies. Eternal immortality. Well, sigh! Unfortunately, humans are useless, although I have come up with a solution. But I have no way to deal with their brains in his mother. Alas, so human civilization is not suitable for me

to use my advanced science and technology in the future,

How to put it, it was a bit too difficult just now, so let's find a simple method. We don't need that method anymore. The above method is too difficult, but essentially the same. This method is definitely very good.

In fact, large-scale verification does not necessarily require large-scale verification, there is another method. The algorithms I created inspired me. Since you're a scanner, you should have heard of them, right?. Literacy refers to the algorithms that make up any scientific machine, the information contained in the algorithms, the information flow carried by the information, the information that can be carried, the natural input that can be carried, the natural scanning that can be carried, and the so-called intelligent machines are only scanned by the information algorithms and feedback information. Since that's the case, large-scale scanning doesn't necessarily require comparison or exclusion. Comparing the living and the dead can also be done through information scanning, direct scanning, and even without the need for so many portrait computer algorithms or intelligent machines. How much time and money do you think this will take. You don't need it, do you. Just the information flow that needs to be loaded.. Is the human brain never dead? Is it practical to learn and apply? That variable undergoes a large-scale scan to identify the missing variable, analyze its components, and determine what can be produced. Once its components are analyzed, there is no way to produce energy. We use energy to produce matter, matter to produce non matter, and antimatter to produce non matter. It's all the same. Since it can be manufactured, you can be there. For example, if a person has already passed away and is missing, then input it to them, and if it is missing, give it to them. What do you think will happen? Also, what would happen if a variable is inputted to a person when their original life form is weakened? If observing and comparing experiments. Do you think I'm not ethical? Or what will we get? That's what I said. Immortal and immortal. From the west to the Qin Emperor, Han Wu constantly pursued Newton throughout history and worked hard all his life. Is this the pursuit?. So, I don't want to die. When and where should I walk and hit the south wall? What if the south wall doesn't work? Let's take a different path, kick the wall or flip over. Isn't this problem solved! What do you think of my ideas and methods. How many years have passed beyond this era.

What we are looking for is the unknown variable. I have always believed that all things in heaven and earth come from reason. So, the only variable among all things in heaven and earth, the only variable that is different between life and death, is the true

soul, or can only be called a soul. Otherwise, what can we call it? What do you think? Life is slow, death is slow. What is the difference between living creatures, plants, animals, and dead creatures in one day, two days, and three days? Just compare and you'll know, what exactly disappeared? Cross comparison and large-scale comparison verification will reveal, spacecraft exploration, but what about the sun? Where's the water? A drop of water, a blade of grass, a flower, even a mountain, a tree, a cloud, a bolt of lightning. As long as we can identify the carrier of cosmic information, we can find out what the variable of everything is? The only variable. Only by deconstructing it and understanding it can it truly comprehend what is the true universe.

Don't worry, life is a long and arduous journey, with plenty of time to test. I never believed that the soul is a neural discharge, and I don't even believe what I say. Science is daring to doubt everything, daring to overthrow everything, daring to verify everything. I even dare to overthrow myself, as long as I am wrong, I live for the truth. Finding the mysteries of all things in the universe is what interests me. Explore it, understand it. Find it. Stream of consciousness and so on, I don't believe in thought. I believe in the unity of mind and matter, where there is thought, there is matter, and everything is relative. then To find that thing, the only variable must be validated against relatively poor ones, even if current people cannot validate it. This is also a core idea. Even if I fail, it cannot be called a failure. It can only be said that the technology of the times is not advanced, but the algorithm I developed has made a step forward, and future people can continue to move forward on my path. Verify black holes and verify galaxies. See if all things in the universe are carriers of information. Looking at the universe, besides matter, is there consciousness? The entire universe is a consciousness body, and all things in the universe are part of the consciousness of the entire universe. A carrier. A flower, a stone, a person, a bird, a sun, a planet, a galaxy black hole, matter antimatter nothingness reality, nothingness reality all together. So, can we measure the true consciousness of the universe?. Can we understand what the universe is thinking from small to large?. So how do we measure it? We can only start small, what if we find it? What is the unique variable contained in all things? Unique variable. So we can communicate with him, explore with him, and deal with that unique variable. Find a way, as long as you can understand it, you can deconstruct it, and deconstruct it naturally. What is the universe really thinking?. What is the thought behind the path we are talking about?.

From philosophical intuition to executable research framework: exploring the scientific path of life and death differences through latent variables

Treating 'soul' as the only verifiable variable is an excellent way to transform philosophical propositions into engineering and experimentation. We can systematically condense this approach into a research system that combines philosophical depth and scientific rigor,

The core proposition is the existence of an observable or inferential latent variable X (i.e., the "unique variable") that exhibits stable differences between "life" and "death", manifested as measurable proxies or patterns in different scales and media; Its testable expression is that for any object of the same kind O (such as a single plant, a single animal tissue, a single drop of water sample), there exists a set of cross modal proxies A=\ {a1, a2, \ dots \} between the "live state" and the "dead state", such that a certain function f (A) statistically significantly distinguishes between the two states,
And maintain consistency and reproducibility across different species, scales, and measurement platforms; The falsifiable condition is that if under strict blind testing, randomization, and control conditions, any candidate f (A) cannot distinguish between life and death under pre-defined efficacy, then the proposition is falsified. At the same time, it is necessary to adhere to philosophical constraints, maintain skepticism andrefutability, and any conclusion must be accompanied by confidence intervals and reproduction conditions. The "soul" should be regarded as the research objective rather than a priori conclusion, and multiple candidates should be tested in parallel.

The priority collection of cross modal proxies covers five categories: firstly, spectroscopic signals, including mass spectrometry, Raman spectroscopy, near-infrared spectroscopy, XRF, etc., which can capture material composition and trace metabolite information;

The second is electrophysiological and chemical indicators, covering microelectrode potential, ion concentration pH、Dissolved oxygen, metabolic rate, etc;

The third is the temporal dynamics characteristics, such as short-term oscillations, noise spectra, and transient recovery curves after light/sound/electrical stimulation;

The fourth is macroscopic physical quantities, including temperature,

transpiration/evaporation rate, reflectance, chlorophyll fluorescence, acoustic characteristics, etc;

The fifth is informative indicators, such as entropy, mutual information, spectral entropy, Lempel Ziv complexity, sample entropy, etc.

In terms of design, three principles must be followed: for the same object, multimodal parallel sampling must be implemented to ensure cross modal alignment. Each device and sample must be equipped with calibration samples and blank controls, and relevant information must be written into the audit Ledger. Sampling must also be repeated at multiple time points such as live, 0 hours, 24 hours, and 72 hours after death to capture dynamic decay patterns.

The experimental research is progressing in an orderly manner in four stages,

Stage 0 is for short-term calibration and synthesis validation, which requires the construction of a noise model and synthesis data generator to generate a control set with known differences and no differences for pipeline validation and threshold tuning. At the same time, a pre registered statistical plan is required to clarify the significance threshold, efficacy, multiple test correction methods, and write them into the audit Ledger.

Phase 1 is a small-scale blind measurement in the short to medium term, selecting individual plants, tissue slices, and small invertebrates as research objects. A double-blind randomized design is adopted to collect multimodal data, train an interpretable model combining sparse VAE and multi view factor models, output candidate latent variables, and conduct blind prediction.

Phase 2 is the mid-term cross scale replication, which involves collaborating with at least 2 to 3 laboratories to independently replicate data from multiple locations, exchanging de identified data and calibration samples, and expanding the research object to include plant community, insect community, microbial community samples, as well as non biological controls such as stones, water droplets, and cloud samples.

Stage 3 is a long-term causal intervention that applies non-destructive interventions such as temperature, humidity, and trace chemicals under controllable conditions, observing the response and reversibility of candidate variable X, in order to advance the construction of the causal evidence chain.

Data analysis and validation require the establishment of a scientific and rigorous methodology system. At the model level, a multi view joint model is used to integrate physiological vectors, material vectors, and temporal vectors for modeling, separating shared and private factors representing candidate X; Using interpretable VAE with

sparse priors and attribution gates, mapping latent variables back to input features facilitates laboratory validation; Using causal toolboxes such as Granger causality test, structural equation modeling, and do calculate approach,

Guide intervention experiment design and causal evidence evaluation. Strictly implement pre registration assumptions, efficacy analysis, and sample size estimation in the statistical process, with blind testing and independent reproduction as core evidence standards, and construct a layered and progressive evidence chain. Starting from observation correlation, complete material spectrum verification, intervention response testing, and cross scale consistency verification in sequence, and set clear reproducibility and auditability standards for each layer.

Engineering assurance is a key support for research rigor and reproducibility, and it is necessary to establish a comprehensive audit Ledger to record the complete metadata and hash values of each sampling, calibration, model training, and triggering event; Create replay and blind test packages, save raw data, process pipelines, random seeds, and model weights for third-party reproduction and research;

Conduct bootstrap stability evaluation on material primitives, eliminate unstable primitives to reduce false positive probability; Deploy real-time drift monitoring mechanism to detect device drift and data distribution drift, and promptly trigger recalibration or manual review processes. At the same time, it is necessary to balance ethics and compliance, and reserve the process and record space for ethics review and environmental impact assessment before subsequent deployment.

From the long-term vision of philosophy and science, even if current technology is not sufficient to fully confirm the existence of a single variable, this algorithm and experimental path are still important advances. It successfully transforms vague philosophical propositions into testable scientific problems. If someone uses higher sensitivity equipment or new measurement methods to reproduce and expand related discoveries in the future, this work will become a key milestone in this research direction. The scientific strategy of small to large, starting from controllable and reproducible small-scale experiments, gradually expanding to macro and cosmic scales, maintaining methodological consistency and reproducibility, will also provide clear direction guidance for subsequent exploration. You can choose synthesis blind test scripts and statistical plans, multi-modal data acquisition and processing pipelines, interpretable VAE and multi view factor model training scripts as needed, and start experiments at any time.

Life and Death Scanner Algorithm

This DeathScanner V10 Ultimate is an industrial grade, parallel multimodal detection and anomaly/"dark residual" discovery platform. The goal is to engineer and extend the original four algorithm ideas I provided (time series modeling, spectral decomposition, anomaly detection, memory/clustering) into an interpretable, auditable, and replayable end-to-end system that supports parallel operation of two paths: edge real-time triggering and cloud offline training.

-

Core module comparison table

Module	Main Responsibilities	Key Algorithms/Technologies
Acquisition layer Sensor	Collect multimodal raw samples and inject metadata	Real driver or Synthetic Sensor rollback
Vectorizers	Mapping physiological and material signals into standardized vectors	Standardization PCA、Differential derivative, spectral preprocessing
MaterialDecomposer	Decompose spectral vectors into elementary coefficients	NMF, elementary library, bootstrap stability
DarkDetector	Detection of Unexplained Residual and Latent Variables	PCA/ICA, IsolationForest, Explainable VAE
VectorIndex/EntRegistry	Real time similarity retrieval and long-term memory/seeding	Faiss or brute force retrieval; Ent/Sed Merge Strategy
Fusion Engine	Cross modal Window Scoring and Triggering Decision	Differential/Variance/Mutual Information Weighted Fusion
Compress and merge Compressor	Merge similar Ents into Seed to reduce redundancy	Cosine similarity greedy clustering
Training and backend Trainer	Material element training, VAE offline training	Asynchronous threading, model registration, Ledger recording
Trigger/Safety	Decision Cooling, Permission and Protection Species Check	Cooling Strategy, Threshold, Permission Management
Monitoring and Drift Monitor	Data Distribution Drift Detection and Alarm	Window Mean/Variance, zscore Detection

Data Flow and Parallel Architecture

1. The collection thread continues to read samples from the sensor (or Synthetic Sensor fallback), puts the samples into the task queue, and writes them to the short-term recent_stamples buffer.
2. Processing Worker Pool Parallel Consumption Queue: Verify Schema and Permissions → Vectorization (Physiology+Materials) → Index Addition → Ent Registration → Window Retrieval → Fusion Scoring → Dark Residual Detection → Ledger Recording and Triggering Decision.
3. Asynchronous execution of material element training and bootstrap stability evaluation by backend training threads; There are also VAE training threads for interpretable latent variable learning (if PyTorch is available).
4. The compression merge cycle runs Compressor regularly to merge similar Ents into Seed and write it to a persistent file.
5. Monitoring and playback: All key events are recorded in the Ledger; Save the sample to the reply directory for blind testing and replication.

Key algorithm details and interpretability design
Vectorization: Baseline/smooth the original physiological channels, concatenate instantaneous derivatives, standardize them, use PCA to reduce dimensionality, and normalize them into unit vectors to ensure stable similarity calculation.
Material decomposition: Learn non negative elements using NMF, output element coefficients, and evaluate element stability through bootstrap; Unstable primitives are labeled as "dark residual" candidates.
Dark residual detection: First, PCA/ICA is used to model the normal subspace, and then covariance/Mahalanobis distance and isolated forest are used to evaluate the residual variability; Optional Explanatable VAE provides latent variables and attribution gates to map latent variables back to input features for experimental validation.
Multimodal fusion: The fusion score is synthesized by weighting the difference between the last frame in the window and the historical mean, channel variance, and cross modal mutual information to trigger the strategy.
Memory and merging: Ent represents the original event cluster, Compressor uses cosine similarity greedy clustering to generate Seed (centroid), and merges the original shards for long-term statistics and playback.
Explanatory outputs: Ledger entries, VAE attribution, material explain fraction, top material bases, etc. are all used as auditable evidence to map "latent variables" back to measurable features.

Advantages, limitations, and engineering support
advantage
End to end engineering: from collection, calibration, indexing, detection to triggering

and auditing the complete chain.
Parallel and scalable: collection thread+multi worker+backend training thread, supporting edge real-time and cloud deep training in parallel.
Explainable and reproducible: VAE attribution, primitive bootstrap, Ledger audit and replay support blind testing and third-party reproduction.
limitation
Dependent on measurement sensitivity: Can the detection of "unique variables" be limited by sensor sensitivity and sampling design.
Statistical efficacy requirements: Pre registered blind testing and sufficient sample size are required to avoid false positives/overfitting.
Ethics and Compliance: Prior to actual deployment, it is necessary to supplement ethical review, environmental impact, and permission management processes.
-Engineering support
Drift detection, model registration, playback and audit Ledger, and permission management are all built-in engineering practices for long-term maintenance and compliance.

Immediate short-term execution: Run a synthetic blind test (using Synthetic Data Generator) to verify end-to-end stability and generate preliminary false positive/false negative statistics.
Mid term: Collect multimodal data using small-scale real blind testing (live vs 0/24/72 hours after death), train ExplainableVAE, and export attribution maps.
Long term: Cross site replication and causal intervention experiments (controllable changes in environmental variables) to test the reversibility and causality of candidate latent variables.

operation log

2026-01-09 00:02:28 [INFO] DeathScannerV10 - No sensor-driver specified; using SyntheticSensor fallback
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Ingest thread started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Worker 0 started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Worker 1 started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Worker 2 started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Worker 3 started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Material trainer thread started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - VAE trainer thread started
2026-01-09 00:02:28 [INFO] DeathScannerV10 - DeathScannerV10UltimateParallel started with 4 workers
2026-01-09 00:02:28 [INFO] DeathScannerV10 - Running until interrupted (Ctrl+C). ..

Real usage methods
Key Attribute Comparison Table
Attributes	Why are they important	Suggested values/practices
Interface contract	Ensure seamless access for different devices	JSON over HTTP/gRPC; Synchronous sampling+asynchronous playback
Dependency and runtime environment	Determine whether it can run on the target machine	Python 3.9+; numpy; Optional Scikit Learn, Torch, Faiss
Deployment mode	Adaptation to resource constraints or cloud training	Edge agent; Container; Cloud service
Sensor driven	Real data source, must be engineered	Clearly read() returns schema and implements calibration/retry
Monitoring and auditing	Observability and reproducibility	Ledger, metrics, replay files, drift alerts

This algorithm is designed as a modular, parallel, and auditable system. The core idea of integrating any machine is to break down the system into three types of portable units: acquisition drivers (Sensors), edge runtime (Agents), and cloud/offline training

and management services. The goal is to enable the minimum runnable unit (edge agent) to run in real-time on resource constrained devices, while asynchronously executing retraining, primitive bootstrap, VAE training, etc. on cloud or stronger nodes.

Preconditions and environmental preparation
Runtime: Python 3.9+virtual environment or container image.
Required library: numpy (mandatory); Optional: scikit learn (PCA/NMF/IF), torch (VAE), Faiss (high-performance indexing).
Hardware: Edge devices need to meet the baseline of memory and CPU; To do VAE training or Faiss GPU, GPU chips and drivers are required.
File system: The writable data/and models/directories are used for Ledger, playback, and model persistence.
Network: If cloud management is adopted, devices need to be able to access management APIs (or report through message queues/proxies).
Permission and Security: Device certificates or API keys used for registration and reporting; Audit Ledger write permission.

Access mode and implementation steps

1. As an edge agent (recommended for on-site devices)
Deployment method: Deploy the merged single file or container image to the device.
Implementation key: Implement the Sensor. read() driver class, which returns JSON (ts, id, meta, physics, material) that conforms to the Data Schema.
Run: Start the Agent, which will start the collection thread, worker pool, and background training thread (if resources allow).
Advantages: low latency triggering, offline availability, data localization.

2. As a containerized microservice (suitable for edge gateways or cloud)
Deployment method: Build Docker images and expose HTTP/gRPC interfaces for receiving samples or queries.
Interface Contract:
POST/input to receive a single sample JSON;
GET/status returns the running status and processed count;
POST/train/material triggers material element training (asynchronous).
Advantages: Easy to expand, unified operation and maintenance, and convenient integration with cloud services.

3. Embed as a library into existing systems (suitable for scientific research or customized platforms)

Deployment method: Import Vectorizer, MaterialDecoder, DarkDetector and other classes as libraries.
Implementation points: Call vectorizer.transform(sample)、index.add(id, vec, meta)、darkdetector.detect(windowvecs)　Waiting for API.
Advantages: Flexible and customizable in depth.

Sensor driver specifications (must be implemented)
Function signature: def read (self) ->Dict [str, Any].
Return schema:
ts: float epoch　秒;
ID: Unique sample ID;
meta:　包含　deviceid, calibversion, permissionid, gps;
Phys: numerical channels (chlf, temp, VOCs [], mass_stec [], etc.);
Material: Spectral array (mass_stec, Raman, SWIR).
Robustness: Implement retry, timeout, exception capture, and local caching (buffering in case of network disconnection).
Calibration: During driver initialization or periodic calls to CalibrationManager. register (device id, version, meta).

Data Flow and API Design
本地流:　Sensor　→　Agent Ingest Queue　→　Worker Pool　→　VectorIndex / EntRegistry　→　Fusion　→　Trigger　→　Ledger。
Report flow (optional): The agent regularly reports the EdgeSummary and Ledger entries to the cloud management service for centralized training and auditing.
Playback mechanism: Each sample is written to the reply/directory for easy blind testing and reproduction.
Model management: After cloud training, the model weights and metadata are written into models/and registered through ModelRegistry, which can be accessed and updated by the agent.

Deployment, Expansion, and Operations

Containerization: Packaging agents as Docker images and managing them using Kubernetes or edge container platforms.
Parallelism tuning: Adjust the number of workers and queue size based on device CPU/memory.
Monitoring: Export Prometheus metrics (processed, queueleng, darkbandidates,

drift_z) and configure alarms.
Log and Audit: Centralize the collection of logs and Ledger, regularly backup and replay data.
Security: TLS, device certificates, minimum privilege principle, tamper proof audit logs (hash chain).
Upgrade strategy: Blue green or rolling upgrade, model updates are first verified in grayscale on small batch devices before being fully distributed.

Testing and Acceptance Process
Unit testing: Boundary behavior testing for Vectorizer, MaterialVectorizer, MaterialDecomposer, DarkDetector.
Integration testing: Use SynthesiziSensor for end-to-end playback to verify queue, index, trigger, and Ledger writes.
Blind testing: Generate synthetic controls and known events to verify false positive/false negative rates.
On site acceptance: Run on the target device for 24-72 hours to check resource utilization, stability, and playback data integrity.

List of Access
1. Implement Sensor driver and ensure the return of DataSchema compliant samples.
2. Select the deployment mode (Edge Agent/Container/Library).
3. Prepare the runtime environment (Python, dependencies, directory permissions, certificates).
4. Start the Agent and observe the processed, Ledger, and reply files.
5. Run synthesis replay to verify end-to-end no abnormalities.
6. Distribute grayscale models and monitor drift and trigger rates.

-

If one day I am willing, I will generate the above access steps into an executable access template: including (a) Sensor driver example code, (b) Dockerfile and Kubernetes deployment examples, (c) HTTP interface specification document, and (d) one click self-test script. I will prepare the template I want first that day.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
deathscannerv10_ultimate.py
DeathScanner V10 Ultimate - Industrial-grade parallel scanning system
Complete single-file implementation with all modules integrated.
"""

from __future__ import annotations
#
================================================================================
==========
# PART 1: CORE INFRASTRUCTURE
#
================================================================================
==========

# Standard library imports
import os
import sys
import time
import json
import math
import uuid
import logging
import threading
import argparse
import queue
import signal
from dataclasses import dataclass, field
```

```
from typing import Any, Dict, List, Optional, Tuple

#
================================================================================
==========
# SECTION 1: DEPENDENCY IMPORTS WITH GRACEFUL FALLBACK
#
================================================================================
==========

# Core numeric libraries - REQUIRED
try:
import numpy as np
numpy_available = True
except ImportError:
print("ERROR: numpy is required. Install with: pip install numpy")
sys.exit(1)

# Scikit-learn - OPTIONAL but recommended
try:
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, FastICA, NMF, FactorAnalysis
from sklearn.ensemble import IsolationForest
from sklearn.covariance import EmpiricalCovariance
sklearn_available = True
except ImportError:
print("WARNING: scikit-learn not available. Some features will be disabled.")
StandardScaler = None
PCA = None
FastICA = None
NMF = None
IsolationForest = None
EmpiricalCovariance = None
FactorAnalysis = None
sklearn_available = False

# FAISS - OPTIONAL for high-performance vector search
try:
import done
faiss_available = True
except ImportError:
faiss_available = False

# PyTorch - OPTIONAL for advanced VAE
```

```
try:
import torch
import torch.nn as nn
import torch.optim as optim
torch_available = True
except ImportError:
torch_available = False

#
================================================================================
==========
# SECTION 2: LOGGING CONFIGURATION
#
================================================================================
==========

log = logging.getLogger("DeathScannerV10")
log.setLevel(logging.INFO)

if not log.handlers:
console_handler = logging. StreamHandler()
console_handler.setLevel(logging.INFO)
formatter = logging. Formatter(
'%(asctime)s [%(levelname)s] %(name)s - %(message)s',
datefmt='%Y-%m-%d %H:%M:%S'
)
console_handler.setFormatter(formatter)
log.addHandler(console_handler)

def now_ts() ->str:
"""Get current timestamp as formatted string."""
return time.strftime("%Y-%m-%d%H:%M:%S", time.localtime())

def uid(prefix: str = "") ->str:
"""Generate a unique identifier with optional prefix."""
return prefix + str(uuid.uuid4())[:12]

def safewritejson(path: str, obj: Any) ->bool:
"""Atomically write JSON object to file."""
try:
tmp_path = path + ".tmp"
with open(tmp_path, "w", encoding="utf-8") as f:
json.dump(obj, f, ensure_ascii=False, indent=2)
os.replace(tmp_path, path)
```

```
return True
except Exception as e:
log.error(f"Failed to write JSON to {path}: {e}")
try:
with open(path, "w", encoding="utf-8") as f:
json.dump(obj, f, ensure_ascii=False, indent=2)
return True
except Exception:
return False

#
==========
==========
# SECTION 3: CONFIGURATION
#
==========
==========

@dataclass
class Config:
"""Global configuration for DeathScanner V10."""

# Directory paths
data_dir: str = "./data"
model_dir: str = "./models"
replay_dir: str = "./data/replay"
calibration_dir: str = "./data/calibration"

# Processing parameters
poll_interval: float = 1.0
window_seconds: float = 1800.0

# Vector dimensions
vector_dim: int = 256
materialembeddim: int = 128

# Index settings
index_type: str = "faiss"
usegpufaiss: bool = False

# Seed/merge parameters
seedmingroup: int = 2
mergesimthreshold: float = 0.88
```

```
# Dark matter detection thresholds
darkresidualthreshold: float = 4.0
darkisolationcontamination: float = 0.005

# Material decomposition
material: int = 12
materialbasisbootstrap: int = 512
materialbootstrapiters: int = 50
materialstabilitythreshold: float = 0.75
materialexplainthreshold: float = 0.35

# VAE settings
enablevaedark: bool = True
Vaelayed: int = 32
vaetrainepochs: int = 200
vaebatchsize: int = 64

# Multi-view factor model
factorshareddim: int = 32

# Safety and permissions
require_permission: bool = True
trigger_confidence: float = 0.88
protectedspecies: List[str] = field(default_factory=list)

def __post_init__(self):
"""Initialize directories and file paths."""
os.makedirs(self.data_dir, exist_ok=True)
os.makedirs(self.model_dir, exist_ok=True)
os.makedirs(self.replay_dir, exist_ok=True)
os.makedirs(self.calibration_dir, exist_ok=True)
self.ledgerpath = os.path.join(self.data_dir, "ledger.json")
self.entspath = os.path.join(self.data_dir, "ents.json")
self.seedspath = os.path.join(self.data_dir, "seeds.json")
self.quarantinepath = os.path.join(self.data_dir, "quarantine.json")

cfg = Config()

# Override from environment variables if present
if os.environ.get('DEATHSCANNER_REQUIRE_PERMISSION') == '0':
cfg.require_permission = False

#
============================================================
```

==========
SECTION 4: LEDGER (AUDIT CHAIN)
#
==========
==========

class Ledger:
"""Audit chain for tracking all system operations."""
_lock = threading. RLock()
chain: List[Dict[str, Any]] = []

@classmethod
def record(cls, op: str, obj_id: str, info: Dict[str, Any]) ->str:
"""Record an operation in ledger."""
with cls. _lock:
prev_hash = cls.chain[-1].get('hash', '') if cls.chain else ''
entry = {
"ts": now_ts(),
"op": op,
"id": obj_id,
"info": info,
"prev": prev_hash
}

# Generate hash
try:
import hashlib
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)

Periodically dump to file for certain operations
if op in ("DARKCANDIDATE", "MATERIALBASISCREATED", "VAETRAINED",
"MATERIAL BASSTABILITY", "MODELREGISTER", "CALIBRATIONS REGISTER,"
"TRIGGER_REPORT", "TRIGGER_VOCS", "EDGEBACKHAULSCHEDULED"):
cls.dump()

return entry['hash']

@classmethod

```
def dump(cls, path: Optional[str] = None) ->bool:
"""Dump ledger to file."""
with cls. _lock:
try:
p = path or cfg.ledgerpath
safewritejson(p, cls.chain)
log.info(f"Ledger dumped to {p} ({len(cls.chain)} entries)")
return True
except Exception as e:
log.error(f"Ledger.dump failed: {e}")
return False

#
==========
# SECTION 5: DATA SCHEMA VALIDATOR
#
==========

DataSchema class:
"""Validator for data samples."""

REQUIRED_TOP_LEVEL = {"ts", "id", "meta", "phys", "material"}
REQUIRED_META = {"deviceid", "calibversion", "permissionid", "gps"}

@staticmethod
def validate(sample: Dict[str, Any]) ->Tuple[bool, str]:
"""Validate a sample against schema."""
if not isinstance(sample, dict):
return False, "sample must be dict"

missing_top = DataSchema. REQUIRED_TOP_LEVEL - set(sample.keys())
if missing_top:
return False, f"missing top-level fields: {missing_top}"

meta = sample.get("meta", {})
if not isinstance(meta, dict):
return False, "meta must be dict"

if cfg.require_permission:
missing_meta = DataSchema. REQUIRED_META - set(meta.keys())
if missing_meta:
return False, f"missing meta fields: {missing_meta}"
```

gps = meta.get("gps", {})
if not isinstance(gps, dict) or ("lat" not in gps or "lon" not in gps):
return False, "meta.gps must include lat and lon"

if not isinstance(sample.get("phys"), dict):
return False, "phys must be dict"

if not isinstance(sample.get("material"), dict):
return False, "material must be dict"

return True, "ok"

#
==========
SECTION 6: PERMISSION AND CALIBRATION MANAGERS
#
==========

class PermissionManager:
"""Manages access permissions."""

def __init__(self):
self.lock = threading. RLock()
self.permissions: Dict[str, Dict[str, Any]] = {}

def register(self, pid: str, info: Dict[str, Any]):
"""Register a new permission."""
with self.lock:
self.permissions[pid] = info
Ledger.record("PERMISSION_REGISTER", pid, info)

def check(self, pid: Optional[str]) ->Tuple[bool, str]:
"""Check if permission is granted."""
if not cfg.require_permission:
return True, "permission_not_required"
if not pid:
return False, "missing_permission"
with self.lock:
if pid not in self.permissions:
return False, "permission_not_found"
return True, "ok"

```
permission_manager = PermissionManager()

class CalibrationManager:
"""Manages device calibration records."""

def __init__(self):
self.lock = threading. RLock()
self.records: Dict[str, Dict[str, Any]] = {}

def register(self, device_id: str, version: str, meta: Dict[str, Any]):
"""Register a device calibration."""
with self.lock:
self.records[device_id] = {
"version": version,
"meta": meta,
"ts": now_ts()
}
Ledger.record("CALIBRATIONREGISTER", uid("cal-"), {
"deviceid": device_id,
"version": version,
"meta": meta
})

def get(self, device_id: str) ->Optional[Dict[str, Any]]:
"""Get calibration for a device."""
return self.records.get(device_id)

calib_mgr = CalibrationManager()

#
============================================================
==========
# SECTION 7: SENSOR INTERFACE
#
============================================================
==========

class Sensor:
"""Abstract sensor interface.

Implement read() in production drivers.
read() must return dict with keys: ts, id, meta, phys, material.
"""
```

```
def read(self) ->Dict[str, Any]:
raise NotImplementedError("Implement Sensor.read() in production driver")

class SyntheticSensor(Sensor):
"""Synthetic sensor for testing and fallback."""

def __init__(self, base_sample: Optional[Dict[str, Any]] = None, rate: float = 1.0):
"""Initialize synthetic sensor.

Args:
base_sample: Template sample to base synthetic data on.
rate: Sampling rate multiplier.
"""
if base_sample is None:
base_sample = {
"ts": time.time(),
"id": uid("synth-"),
"meta": {
"deviceid": "synth-0",
"calibversion": "v1",
"permission_id": "perm-synth",
"gps": {"lat": 0.0, "lon": 0.0}
},
"phys": {
"chlf": 0.1,
"temp": 20.0,
"acoustic": 0.0,
"microelectrode": 0.0,
"vocs": [0.0] * 8,
"mass_spec": [0.0] * 16
},
"material": {
"mass_spec": [0.0] * 64,
"raman": [0.0] * 128,
"swir": [0.0] * 64
}
}
self.base = base_sample
self.rate = rate
self.counter = 0

def read(self) ->Dict[str, Any]:
"""Generate a synthetic sample."""
self.counter += 1
```

```
s = json.loads(json.dumps(self.base))
s["ts"] = time.time()
s["id"] = uid("synth-")

# Perturb phys
phys = s["phys"]
phys["temp"] = float(phys.get("temp", 20.0) + np.random.normal(scale=0.1))
phys["chlf"] = float(max(0.0, phys.get("chlf", 0.1) + np.random.normal(scale=0.01)))
phys["acoustic"] = float(phys.get("acoustic", 0.0) + np.random.normal(scale=0.05))

# Add random mass peaks
mat = s["material"]
mat["mass_spec"] = [float(x + np.random.normal(scale=0.001)) for x in
mat.get("mass_spec", [])]
mat["raman"] = [float(x + np.random.normal(scale=0.002)) for x in mat.get("raman", [])]
mat["swir"] = [float(x + np.random.normal(scale=0.001)) for x in mat.get("swir", [])]

s["phys"] = phys
s["material"] = mat

return s

#
==========
==========
# SECTION 8: VECTORIZERS
#
==========
==========

class Vectorizer:
"""Vectorizes physiological data."""

def __init__(self, dim: int = cfg.vector_dim):
self.dim = dim
self.scaler = StandardScaler() if sklearn_available else None
self.pca = PCA(n_components=min(64, dim)) if sklearn_available else None
self. _warmup = False
self. _lock = threading. RLock()

def basicfeatures(self, sample: Dict[str, Any]) ->np.ndarray:
"""Extract basic features from sample."""
phys = sample.get("phys", {})
chlf = float(phys.get("chlf") or 0.0)
```

```
temp = float(phys.get("temp") or 0.0)
acoustic = float(phys.get("acoustic") or 0.0)
micro = float(phys.get("microelectrode") or 0.0)
vocs = phys.get("vocs") or []
vocsfixed = (vocs + [0.0] * 24)[:24]
mass = phys.get("mass_spec") or []
massfixed = (mass + [0.0] * 32)[:32]

arr = np.array([chlf, temp, acoustic, micro] + vocsfixed + massfixed, dtype=float)
return arr

def fit_warmup(self, samples: List[Dict[str, Any]]):
"""Warmup vectorizer with samples."""
with self. _lock:
mats = [self.basicfeatures(s) for s in samples]
X = np.stack(mats, axis=0)

if self.scaler:
self.scaler.fit(X)
Xs = self.scaler.transform(X)
else:
Xs = X

if self.pca:
self.pca.fit(Xs)

self. _warmup = True
log.info(f"Vectorizer warmup done on {len(samples)} samples")

def transform(self, sample: Dict[str, Any], window: Optional[List[Dict[str, Any]]] =
None) ->np.ndarray:
"""Transform sample to vector."""
with self. _lock:
base = self.basicfeatures(sample)
deriv = np.zeros_like(base)

if window and len(window)>= 2:
prev = np.stack([self.basicfeatures(s) for s in window[:-1]], axis=0)
prevmean = np.mean(prev, axis=0)
deriv = base - prevmean

vecraw = np.concatenate([base, deriv], axis=0)

if self.scaler and self. _warmup:
```

```
try:
vecscaled = self.scaler.transform(vecraw.reshape(1, -1))[0]
except Exception:
vecscaled = vecraw
else:
vecscaled = vecraw

if self.pca and self. _warmup:
try:
vecp = self.pca.transform(vecscaled.reshape(1, -1))[0]
except Exception:
vecp = vecscaled[:self.pca.n_components]
else:
vecp = vecscaled

if len(vecp)>= self.dim:
emb = vecp[:self.dim]
else:
emb = np.concatenate([vecp, np.zeros(self.dim - len(vecp))], axis=0)

norm = np.linalg.norm(emb) + 1e-12
emb = emb / norm
return emb.astype(float)

class MaterialVectorizer:
"""Vectorizes material data."""

def __init__(self, masslen: int = 64, ramanlen: int = 256, swirlen: int = 128,
embeddim: int = cfg.materialembeddim):
self.masslen = masslen
self.ramanlen = ramanlen
self.swirlen = swirlen
self.embeddim = embeddim
self.scaler = StandardScaler() if sklearn_available else None
self. _warmup = False
self. _lock = threading. RLock()

def preprocess(self, spec: List[float], targetlen: int) ->np.ndarray:
"""Preprocess spectral data."""
arr = np.array((spec or [])[:targetlen], dtype=float)
if arr.size<targetlen:
arr = np.concatenate([arr, np.zeros(targetlen - arr.size)])
arr = arr - np.median(arr)
if arr.size>= 3:
```

```
kernel = np.ones(3) / 3.0
arr = np.convolve(arr, kernel, mode='same')
norm = np.linalg.norm(arr) + 1e-12
return (arr / norm).astype(float)

def fit_warmup(self, samples: List[Dict[str, Any]]):
"""Warmup with samples."""
with self. _lock:
mats = []
for s in samples:
mat = s.get("material", {}) or {}
mass = self.preprocess(mat.get("mass_spec") or [], self.masslen)
raman = self.preprocess(mat.get("raman") or [], self.ramanlen)
swir = self.preprocess(mat.get("swir") or [], self.swirlen)
vec = np.concatenate([mass, raman, swir], axis=0)
mats.append(vec)

if not mats:
return

X = np.stack(mats, axis=0)
if self.scaler:
self.scaler.fit(X)

self. _warmup = True
log.info(f"MaterialVectorizer warmup done on {len(samples)} samples")

def transform(self, sample: Dict[str, Any]) ->np.ndarray:
"""Transform sample to material vector."""
with self. _lock:
mat = sample.get("material", {}) or {}
mass = self.preprocess(mat.get("mass_spec") or [], self.masslen)
raman = self.preprocess(mat.get("raman") or [], self.ramanlen)
swir = self.preprocess(mat.get("swir") or [], self.swirlen)
vec = np.concatenate([mass, raman, swir], axis=0)

if self.scaler and self. _warmup:
try:
vec = self.scaler.transform(vec.reshape(1, -1))[0]
except Exception:
pass

if vec.size>= self.embeddim:
emb = vec[:self.embeddim]
```

```
else:
emb = np.concatenate([vec, np.zeros(self.embeddim - vec.size)], axis=0)

norm = np.linalg.norm(emb) + 1e-12
emb = emb / norm
return emb.astype(float)

#
==========
# SECTION 9: MATERIAL DECOMPOSITION
#
==========

class MaterialDecomposer:
"""Decomposes material vectors into basis components."""

def __init__(self, nbases: int = cfg.materialbasisk):
self.nbases = nbases
self.model = NMF(n_components=self.nbases, init='nndsvda', max_iter=1000) if
sklearn_available else None
self.basis = None
self. _trained = False
self. _lock = threading. RLock()

def fitbasis(self, materialmatrix: np.ndarray, bootstrap: bool = False):
"""Train material basis."""
with self. _lock:
if self.model is None:
log.warning("NMF not available; material basis training skipped")
return

try:
W = self.model.fit_transform(np.abs(materialmatrix) + 1e-12)
H = self.model.components_
self.basis = H
self. _trained = True
Ledger.record("MATERIALBASISCREATED", uid("matbasis-"), {
"nbases": self.nbases,
"samples": materialmatrix.shape[0]
})
log.info(f"Material basis trained shape={str(self.basis.shape)}")
except Exception as e:
```

```
log.error(f"MaterialDecomposer.fit_basis failed: {e}")
self. _trained = False

def decompose(self, material_vec: np.ndarray) ->List[float]:
"""Decompose material vector into coefficients."""
with self. _lock:
if self.model is None or not self. _trained:
if self.basis is not None:
coeffs = np.maximum(0.0, np.dot(self.basis, material_vec))
s = np.sum(coeffs) + 1e-12
return (coeffs / s).tolist()
return [0.0] * (self.nbases or 1)

try:
coeffs = self.model.transform(np.abs(material_vec).reshape(1, -1))[0]
s = np.sum(coeffs) + 1e-12
return (coeffs / s).tolist()
except Exception as e:
log.error(f"MaterialDecomposer.decompose failed: {e}")
return [0.0] * (self.nbases or 1)

class MaterialBasisStability:
"""Evaluates stability of material basis using bootstrap."""

def __init__(self, decomposer: MaterialDecomposer):
self.decomposer = decomposer

def bootstrapstability(self, materialmatrix: np.ndarray, niter: int =
cfg.materialbootstrapiters) ->Dict[str, Any]:
"""Perform bootstrap stability assessment."""
if not sklearn_available:
return {"status": "skipped", "reason": "scikit-learn not available"}

bases = []
n = materialmatrix.shape[0]

for i in range(niter):
idx = np.random.choice(n, size=n, replace=True)
try:
model = NMF(n_components=self.decomposer.nbases, init='nndsvda', max_iter=500)
W = model.fit_transform(np.abs(materialmatrix[idx]) + 1e-12)
H = model.components_
bases.append(H)
except Exception:
```

```
continue

if not bases:
return {"status": "failed", "reason": "no bases generated"}

ref = bases[0]
scores = []
for b in bases[1:]:
sim = np.abs(np.dot(ref, b.T))
max_per_ref = np.max(sim, axis=1)
scores.append(float(np.mean(max_per_ref)))

stability = float(np.mean(scores)) if scores else 0.0
Ledger.record("MATERIALBASISSTABILITY", uid("mbs-"), {
"stability": stability,
"samples": materialmatrix.shape[0]
})
return {"status": "ok", "stability": stability, "bootstrap_iters": len(bases)}

#
==========
# SECTION 10: VECTOR INDEX
#
==========

class VectorIndex:
"""High-performance vector similarity search."""

def __init__(self, dim: int = cfg.vector_dim):
self.dim = dim
self.lock = threading. RLock()
self.idtometa: Dict[int, Dict[str, Any]] = {}
self.idmap: Dict[int, str] = {}
self.revidmap: Dict[str, int] = {}
self.next_idx = 0
self.vectors: List[np.ndarray] = []
self.faiss_index = None

if cfg.index_type == "faiss" and faiss_available:
try:
if cfg.usegpufaiss and hasattr(faiss, 'StandardGpuResources'):
res = faiss. StandardGpuResources()
```

```
idx = faiss. IndexFlatIP(self.dim)
self.faiss_index = faiss.index_cpu_to_gpu(res, 0, idx)
else:
self.faiss_index = faiss. IndexFlatIP(self.dim)
log.info(f"Faiss index initialized dim={self.dim}")
except Exception as e:
log.warning(f"Faiss init failed; falling back to brute-force: {e}")
self.faiss_index = None

def add(self, uid_str: str, vec: np.ndarray, meta: Dict[str, Any]):
"""Add vector to index."""
with self.lock:
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())}

if self.faiss_index is not None:
v = np.array(vec, dtype='float32').reshape(1, -1)
self.faiss_index.add(v)
else:
self.vectors.append(np.array(vec, dtype=float))

Ledger.record("INDEXADD", uid_str, {"idx": idx, "ts": meta.get("ts")})

def query(self, vec: np.ndarray, topk: int = 10) ->List[Dict[str, Any]]:
"""Query for similar vectors."""
with self.lock:
if self.faiss_index is not None:
v = np.array(vec, dtype='float32').reshape(1, -1)
D, I = self.faiss_index.search(v, topk)
res = []
for score, idx in zip(D[0], I[0]):
if idx<0:
continue
uid_str = self.idmap.get(int(idx))
res.append({
"id": uid_str,
"score": float(score),
"meta": self.idtometa.get(int(idx))
})
return res
else:
```

```
if not self.vectors:
return []
mats = np.stack(self.vectors, axis=0)
v = vec.reshape(1, -1)
sims = (mats @ v.T).reshape(-1)
idxs = np.argsort(-sims)[:topk]
res = []
for i in idxs:
uid_str = self.idmap.get(i)
res.append({
"id": uid_str,
"score": float(sims[i]),
"meta": self.idtometa.get(i)
})
return res

def queryrecent(self, ts: float, windowseconds: float = cfg.window_seconds,
topk: int = 50) ->List[Dict[str, Any]]:
"""Query for recent vectors."""
with self.lock:
cutoff = ts - windowseconds
ids = [i for i, m in self.idtometa.items() if m.get("ts", 0)>= cutoff]
ids_sorted = sorted(ids, key=lambda x: self.idtometa[x].get("ts", 0),
reverse=True)[:topk]
return [{"id": self.id_map[i], "meta": self.idtometa[i]} for i in ids_sorted]

#
==========
==========
# SECTION 11: MEMORY - ENT/SEED REGISTRY
#
==========
==========

@dataclass
class EntNode:
"""Entity node representing a cluster of data."""
id: str
vec: Optional[List[float]]
shards: List[str]
score: float = 1.0
ts: float = field(default_factory=time.time)

def to_dict(self) ->Dict[str, Any]:
```

```
return {
"id": self.id,
"vec": self.vec,
"shards": self.shards,
"score": self.score,
"ts": self.ts
}

@staticmethod
def from_dict(d: Dict[str, Any]) ->'EntNode':
return EntNode(
id=d["id"],
vec=d.get("vec"),
shards=d.get("shards", []),
score=d.get("score", 1.0),
ts=d.get("ts", time.time())
)

@dataclass
class SeedNode:
"""Seed node representing a cluster centroid."""
id: str
seed_vec: Optional[List[float]]
members: List[str]
diffs: Dict[str, List[int]] = field(default_factory=dict)
quantmeta: Dict[str, Any] = field(default_factory=dict)
ts: float = field(default_factory=time.time)

def to_dict(self) ->Dict[str, Any]:
return {
"id": self.id,
"seed_vec": self.seed_vec,
"members": self.members,
"diffs": self.diffs,
"quant_meta": self.quantmeta,
"ts": self.ts
}

@staticmethod
def from_dict(d: Dict[str, Any]) ->'SeedNode':
return SeedNode(
id=d["id"],
seed_vec=d.get("seed_vec"),
members=d.get("members", []),
```

```
diffs=d.get("diffs", {}),
quantmeta=d.get("quant_meta", {}),
ts=d.get("ts", time.time())
)

class EntRegistry:
"""Registry of entity nodes."""

def __init__(self, path: str = None):
self.path = path or cfg.entspath
self.lock = threading. RLock()
self.nodes: Dict[str, EntNode] = {}
self. _load()

def _load(self):
"""Load registry from file."""
if os.path.exists(self.path):
try:
with open(self.path, "r", encoding="utf-8") as f:
data = json.load(f)
for nid, nd in data.get("nodes", {}).items():
self.nodes[nid] = EntNode.from_dict(nd)
log.info(f"Loaded {len(self.nodes)} ents")
except Exception as e:
log.error(f"EntRegistry load failed: {e}")
self.nodes = {}

def save(self):
"""Save registry to file."""
with self.lock:
data = {"nodes": {nid: n.to_dict() for nid, n in self.nodes.items()}}
safewritejson(self.path, data)

def register(self, node: EntNode):
"""Register a new entity node."""
with self.lock:
self.nodes[node.id] = node
try:
safewritejson(self.path, {
"nodes": {nid: n.to_dict() for nid, n in self.nodes.items()}
})
except Exception:
pass
Ledger.record("ENT_REGISTER", node.id, {"shards": len(node.shards)})
```

```
class SeedIndex:
"""Index of seed nodes."""

def __init__(self, path: str = None):
self.path = path or cfg.seedspath
self.lock = threading. RLock()
self.seeds: Dict[str, SeedNode] = {}
self. _load()

def _load(self):
"""Load index from file."""
if os.path.exists(self.path):
try:
with open(self.path, "r", encoding="utf-8") as f:
data = json.load(f)
for sid, sd in data.get("seeds", {}).items():
self.seeds[sid] = SeedNode.from_dict(sd)
log.info(f"Loaded {len(self.seeds)} seeds")
except Exception as e:
log.error(f"SeedIndex load failed: {e}")
self.seeds = {}

def save(self):
"""Save index to file."""
with self.lock:
data = {"seeds": {sid: s.to_dict() for sid, s in self.seeds.items()}}
safewritejson(self.path, data)

def register(self, seed: SeedNode):
"""Register a new seed node."""
with self.lock:
self.seeds[seed.id] = seed
try:
safewritejson(self.path, {
"seeds": {sid: s.to_dict() for sid, s in self.seeds.items()}
})
except Exception:
pass
Ledger.record("SEED_REGISTER", seed.id, {"members": len(seed.members)})

#
==========
```

SECTION 12: COMPRESSOR / MERGE LOGIC
#
==========
==========

def cosine(a: List[float], b: List[float]) ->float:
"""Calculate cosine similarity between two vectors."""
if a is None or b is None:
return 0.0
aa = np.array(a, dtype=float)
bb = np.array(b, dtype=float)
an = np.linalg.norm(aa) + 1e-12
bn = np.linalg.norm(bb) + 1e-12
return float(np.dot(aa, bb) / (an * bn))

class Compressor:
"""Compresses entity nodes into seeds."""

def __init__(self, registry: EntRegistry, seed_index: SeedIndex):
self.registry = registry
self.seedindex = seed_index

def greedycluster(self, nodes: List[EntNode], simthresh: float,
min_group: int) ->List[List[EntNode]]:
"""Greedy clustering of similar nodes."""
groups = []
used = set()

for i, a in enumerate(nodes):
if a.id in used:
continue
group = [a]
used.add(a.id)

for b in nodes[i+1:]:
if b.id in used:
continue
try:
if cosine(a.vec, b.vec)>= simthresh:
group.append(b)
used.add(b.id)
except Exception:
continue

```
if len(group)>= min_group:
groups.append(group)

return groups

def buildseeds(self, simthresh: float = cfg.mergesimthreshold,
mingroup: int = cfg.seedmingroup) ->Dict[str, int]:
"""Build seeds from entities."""
nodes = list(self.registry.nodes.values())
if not nodes:
return {"created": 0}

groups = self.greedycluster(nodes, simthresh, mingroup)
created = 0

for g in groups:
created += self.createseed(g)

return {"created": created, "groups": len(groups)}

def createseed(self, group: List[EntNode]) ->int:
"""Create a seed from a group of entities."""
vecs = [n.vec for n in group if n.vec is not None]
if not vecs:
return 0

seed_vec = np.mean(np.stack(vecs, axis=0), axis=0).tolist()
memberids = []
diffs = {}

for n in group:
memberids.extend(n.shards)
if n.vec is not None:
d = (np.array(n.vec) - np.array(seed_vec)).tolist()
diffs[n.shards[0] if n.shards else n.id] = [
int(round(x * 1000)) for x in d[:min(64, len(d))]
]

seednode = SeedNode(
id=uid("seed-"),
seed_vec=seed_vec,
members=memberids,
diffs=diffs,
quantmeta={"method": "mean"}
```

```
)

self.seedindex.register(seednode)

with self.registry.lock:
for n in group:
self.registry.nodes.pop(n.id, None)

mergedent = EntNode(
id=uid("ent-"),
vec=seed_vec,
shards=memberids,
score=sum(n.score for n in group)
)
self.registry.nodes[mergedent.id] = mergedent
try:
self.registry.save()
except Exception:
pass

Ledger.record("SEEDCREATED", seednode.id, {
"from": [n.id for n in group],
"members": len(memberids)
})
log.info(f"Created seed {seednode.id} from {len(group)} ents")
return 1

#
==========================================================================
==========
# SECTION 13: FUSION ENGINE
#
==========================================================================
==========

def computemutualinfo(a: np.ndarray, b: np.ndarray) ->float:
"""Compute mutual information proxy (correlation)."""
try:
if a.size and b.size:
return float(np.corrcoef(a.flatten(), b.flatten())[0, 1])
except Exception:
pass
return 0.0
```

```
class FusionEngine:
"""Fuses multiple signals into a unified score."""

def __init__(self):
pass

def scorewindow(self, windowvecs: List[np.ndarray],
metas: List[Dict[str, Any]]) ->Tuple[float, Dict[str, float]]:
"""Compute fusion score over a window of vectors."""
if not windowvecs:
return 0.0, {}

arr = np.stack(windowvecs, axis=0)
last = arr[-1]
mean = np.mean(arr[:-1], axis=0) if arr.shape[0]>1 else arr[0]

diff = np.linalg.norm(last - mean) / (np.linalg.norm(mean) + 1e-12)
var = float(np.mean(np.var(arr, axis=0)))

mi = 0.0
if metas and len(metas)>= 2:
try:
chlf_series = np.array([m.get("meta", {}).get("chlf", 0.0) for m in metas])
temp_series = np.array([m.get("meta", {}).get("temp", 0.0) for m in metas])
mi = computemutualinfo(chlf_series, temp_series)
except Exception:
mi = 0.0

fused = 0.6 * (1.0 - math.exp(-6.0 * diff)) + \
0.3 * (1.0 - math.exp(-2.0 * var)) + \
0.1 * abs(mi)
fused = max(0.0, min(1.0, fused))

top_features = {"diff": float(diff), "var": float(var), "mi": float(mi)}
return fused, top_features

#
==========
# PART 2: ADVANCED FEATURES AND PARALLEL ARCHITECTURE
#
==========
```

#
==========
SECTION 14: EXPLAINABLE VAE
#
==========

if torch_available:
class ExplainableVAE(nn.Module):
"""Explainable Variational Autoencoder with attention gating."""

def __init__(self, inputdim: int, latentdim: int = cfg.vaelatentdim,
hidden: int = 512, sparse_lambda: float = 1e-4):
super(). __init__()
self.inputdim = inputdim
self.latentdim = latentdim
self.sparselambda = sparse_lambda

self.encfc1 = nn. Linear(inputdim, hidden)
self.encfc2 = nn. Linear(hidden, hidden // 2)
self.mu = nn. Linear(hidden // 2, latentdim)
self.logvar = nn. Linear(hidden // 2, latentdim)

self.decfc1 = nn. Linear(latentdim, hidden // 2)
self.decfc2 = nn. Linear(hidden // 2, hidden)
self.out = nn. Linear(hidden, inputdim)

self.attn = nn. Linear(inputdim, inputdim)
self.act = nn. ReLU()

def encode(self, x: torch.Tensor) ->Tuple[torch.Tensor, torch.Tensor]:
"""Encode input to latent parameters."""
h = self.act(self.encfc1(x))
h = self.act(self.encfc2(h))
return self.mu(h), self.logvar(h)

def reparam(self, mu: torch.Tensor, logvar: torch.Tensor) ->torch. Tensor:
"""Reparameterization trick."""
std = torch.exp(0.5 * logvar)
eps = torch.randn_like(std)
return mu + eps * std

def decode(self, z: torch.Tensor) ->torch. Tensor:

"""Decode latent to output."""
h = self.act(self.decfc1(z))
h = self.act(self.decfc2(h))
return self.out(h)

def forward(self, x: torch.Tensor) ->Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]:
"""Forward pass."""
mu, logvar = self.encode(x)
z = self.reparam(mu, logvar)
recon = self.decode(z)
gate = torch.sigmoid(self.attn(x))
recon = recon * gate
return recon, mu, logvar, gate

def trainvaetorch(model: ExplainableVAE, X: np.ndarray,
epochs: int = cfg.vaetrainepochs, batchsize: int = cfg.vaebatchsize,
lr: float = 1e-3, savepath: Optional[str] = None) ->bool:
"""Train VAE with PyTorch."""
try:
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
optimizer = optim. Adam(model.parameters(), lr=lr)

data = torch.tensor(X, dtype=torch.float32).to(device)
ds = torch.utils.data. TensorDataset(data)
loader = torch.utils.data. DataLoader(ds, batch_size=batchsize, shuffle=True)

model.train()
for epoch in range(epochs):
total_loss = 0.0
for batch in loader:
x = batch[0]
recon, mu, logvar, gate = model(x)

reconloss = nn.functional.mse_loss(recon, x, reduction='sum') / x.size(0)
kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) / x.size(0)
sparsity = torch.mean(torch.abs(gate))
loss = reconloss + 1e-3 * kld + model.sparselambda * sparsity

optimizer.zero_grad()
loss.backward()
optimizer.step()
total_loss += loss.item()

```
if epoch % 10 == 0:
log.info(f"VAE epoch {epoch} loss {total_loss / max(1, len(loader)):.6f}")

if savepath:
torch.save(model.state_dict(), savepath)
Ledger.record("VAETRAINED", uid("vae-"), {
"epochs": epochs,
"samples": X.shape[0],
"path": savepath
})
return True
except Exception as e:
log.error(f"VAE training failed: {e}")
return False
else:
class ExplainableVAE:
"""Fallback VAE using PCA."""

def __init__(self, inputdim: int, latentdim: int = cfg.vaelatentdim):
self.inputdim = inputdim
self.latentdim = latentdim
self.mean = None
self.components = None

def fit(self, X: np.ndarray):
"""Fit PCA."""
self.mean = np.mean(X, axis=0)
Xc = X - self.mean
U, S, Vt = np.linalg.svd(Xc, full_matrices=False)
self.components = Vt[:self.latentdim]
Ledger.record("VAEFALLBACKTRAINED", uid("vae-"), {"samples": X.shape[0]})

def encode(self, x: np.ndarray) ->np.ndarray:
"""Encode to latent."""
if self.components is None:
raise RuntimeError("Fallback VAE not trained")
return np.dot(x - self.mean, self.components.T)

def decode(self, z: np.ndarray) ->np.ndarray:
"""Decode from latent."""
return np.dot(z, self.components) + self.mean

def attribution(self, x: np.ndarray) ->List[float]:
```

"""Compute attribution."""
return np.abs(x - self.mean).tolist()

#
==========
SECTION 15: MULTI-VIEW FACTOR MODEL
#
==========

class MultiViewFactorModel:
"""Multi-view factor analysis model."""

def __init__(self, shareddim: int = cfg.factorshareddim):
self.shareddim = shareddim
self.fa = FactorAnalysis(n_components=shareddim) if sklearn_available else None
self.scalers = {}
self. _trained = False

def fit(self, views: Dict[str, np.ndarray]):
"""Fit model on multiple views."""
Xs = []
for k, v in views.items():
if sklearn_available and StandardScaler is not None:
s = StandardScaler()
vs = s.fit_transform(v)
self.scalers[k] = s
else:
vs = v
Xs.append(vs)

X = np.concatenate(Xs, axis=1)
if self.fa is not None:
self.fa.fit(X)
self. _trained = True
Ledger.record("MULTIVIEWFACTORTRAINED", uid("mvf-"), {
"shareddim": self.shareddim,
"views": list(views.keys()),
"samples": X.shape[0]
})

def transform(self, views: Dict[str, np.ndarray]) ->np.ndarray:
"""Transform views to shared space."""

```
if not self. _trained:
raise RuntimeError("MultiViewFactorModel not trained")

Xs = []
for k, v in views.items():
s = self.scalers.get(k)
if s:
Xs.append(s.transform(v))
else:
Xs.append(v)

X = np.concatenate(Xs, axis=1)
return self.fa.transform(X)

#
==========
# SECTION 16: NOISE SIMULATOR
#
==========

class NoiseSimulator:
"""Simulate various types of noise."""

def __init__(self, seed: int = 42):
self.rng = np.random.default_rng(seed)

def gaussian_noise(self, vec: np.ndarray, snr: float = 20.0) ->np.ndarray:
"""Add Gaussian noise."""
power = np.mean(vec ** 2) + 1e-12
sigma = math.sqrt(power) / (snr + 1e-12)
return vec + self.rng.normal(scale=sigma, size=vec.shape)

def multiplicative_noise(self, vec: np.ndarray, scale: float = 0.01) ->np.ndarray:
"""Add multiplicative noise."""
return vec * (1.0 + self.rng.normal(scale=scale, size=vec.shape))

def drift(self, vec: np.ndarray, magnitude: float = 0.01) ->np.ndarray:
"""Add cumulative drift."""
drift_vec = magnitude * (self.rng.normal(size=vec.shape).cumsum() /
(np.arange(len(vec)) + 1))
return vec + drift_vec
```

```
#
==========
# SECTION 17: SYNTHETIC DATA GENERATOR
#
==========

class SyntheticDataGenerator:
"""Generate synthetic data from templates."""

def __init__(self, vectorizer: Vectorizer, materialvectorizer: MaterialVectorizer,
noisesim: NoiseSimulator):
self.vectorizer = vectorizer
self.materialvectorizer = materialvectorizer
self.noisesim = noisesim

def synthfromsample(self, sample: Dict[str, Any], n_variants: int = 10,
perturb: float = 0.05) ->List[Dict[str, Any]]:
"""Generate synthetic variants of a sample."""
variants = []
for i in range(n_variants):
s = json.loads(json.dumps(sample))
phys = s.get("phys", {})
for k, v in list(phys.items()):
if isinstance(v, (int, float)):
phys[k] = float(v * (1.0 + np.random.normal(scale=perturb)))
elif isinstance(v, list):
phys[k] = [float(x * (1.0 + np.random.normal(scale=perturb))) for x in v]

mat = s.get("material", {})
for k, v in list(mat.items()):
if isinstance(v, list):
mat[k] = [float(x * (1.0 + np.random.normal(scale=perturb))) for x in v]

s["phys"] = phys
s["material"] = mat
s["id"] = uid("synth-")
s["ts"] = time.time()
variants.append(s)
return variants

#
==========
```

==========
SECTION 18: MONITOR AND DRIFT DETECTION
#
==========
==========

class Monitor:
"""Monitor statistics and detect drift."""
def __init__(self):
self.lock = threading. RLock()
self.stats: List[Dict[str, Any]] = []

def record(self, X: np.ndarray):
"""Record statistics for vector."""
with self.lock:
self.stats.append({
"ts": now_ts(),
"mean": float(np.mean(X)),
"std": float(np.std(X))
})
if len(self.stats)>1000:
self.stats.pop(0)

def detectdrift(self, window: int = 50, zthreshold: float = 3.0) ->Tuple[bool, float]:
"""Detect if data has drifted."""
with self.lock:
if len(self.stats)<window:
return False, 0.0

means = np.array([s["mean"] for s in self.stats[-window:]])
z = (means[-1] - means.mean()) / (means.std() + 1e-12)

if abs(z)>zthreshold:
Ledger.record("DATA_DRIFT", uid("drift-"), {"z": float(z)})
return True, float(z)
return False, float(z)

monitor = Monitor()

#
==========
==========
SECTION 19: TRIGGER ENGINE

```
#
==========

class Trigger:
"""Evaluate triggers based on fusion scores."""

def __init__(self, confthreshold: float = cfg.trigger_confidence):
self.confthreshold = confthreshold
self.cooldowns: Dict[str, float] = {}

def _allowed(self, action: str) ->bool:
"""Check if action is allowed (not on cooldown)."""
now = time.time()
return now>= self.cooldowns.get(action, 0.0)

def setcd(self, action: str, seconds: float):
"""Set cooldown for action."""
self.cooldowns[action] = time.time() + seconds

def evaluate(self, fusedscore: float, darkinfo: Dict[str, Any],
meta: Dict[str, Any]) ->Tuple[str, Dict[str, Any]]:
"""Evaluate if trigger should fire."""
if darkinfo.get("isanomaly") and \
darkinfo.get("materialexplainfraction", 0.0)<cfg.materialexplainthreshold and \
self. _allowed("dark_report"):
self.setcd("dark_report", 600.0)
return "dark_report", {"reason": "dark_unexplained", "z": darkinfo.get("zscore")}

if fusedscore>= 0.995 and self. _allowed("report"):
self.setcd("report", 300.0)
return "report", {"reason": "very_high_confidence", "score": fusedscore}

if fusedscore>= self.confthreshold and self. _allowed("vocs_sample"):
self.setcd("vocs_sample", 120.0)
return "vocs_sample", {"reason": "high_confidence", "score": fusedscore}

if fusedscore>= (self.confthreshold * 0.7) and self. _allowed("repoll"):
self.setcd("repoll", 30.0)
return "repoll", {"reason": "medium_confidence", "score": fusedscore}

return "noop", {}

#
```

```
# ============================================================
# SECTION 20: SAFETY ENGINE
# ============================================================

class SafetyEngine:
    """Ensure safe operation."""

    def __init__(self, protectedspecies: Optional[List[str]] = None):
        self.protected = set(protectedspecies or [])

    def check(self, sample_meta: Dict[str, Any]) ->Tuple[bool, str]:
        """Check if sample is safe to process."""
        species = samplemeta.get("meta", {}).get("speciesguess")
        permission = samplemeta.get("meta", {}).get("permissionid")

        if species and species in self.protected:
            return False, "protected_species"

        if cfg.require_permission and not permission:
            return False, "missing_permission"

        return True, "ok"

# ============================================================
# SECTION 21: EDGE SUMMARY
# ============================================================

class EdgeSummary:
    """Generate edge summaries for transmission."""

    def __init__(self, vectorizer: Vectorizer, darkdetector: 'DarkMatterDetector'):
        self.vectorizer = vectorizer
        self.darkdetector = darkdetector

    def summarize(self, sample: Dict[str, Any], window: List[Dict[str, Any]]) ->Dict[str, Any]:
        """Summarize sample."""
```

```
vec = self.vectorizer.transform(sample, window=window)
arr = np.stack([self.vectorizer.transform(s, window=window) for s in window], axis=0)
if window else np.stack([vec], axis=0)
mean = np.mean(arr, axis=0)
dist = float(np.linalg.norm(vec - mean))
darkinfo = self.darkdetector.detect(arr.tolist(), None) if hasattr(self.darkdetector,
"detect") else {"score": 0.0, "zscore": 0.0}
return {
"id": sample.get("id"),
"ts": sample.get("ts"),
"embed": vec.tolist(),
"dist": dist,
"darkscore": darkinfo.get("score"),
"darkz": darkinfo.get("zscore")
}

#
==========
# SECTION 22: EXPERIMENT RUNNER
#
==========

class ExperimentRunner:
"""Manage experiments."""

def __init__(self):
self.lock = threading. RLock()
self.experiments: Dict[str, Dict[str, Any]] = {}

def register(self, exp_id: str, design: Dict[str, Any]):
"""Register a new experiment."""
with self.lock:
self.experiments[exp_id] = design
Ledger.record("EXPERIMENTREGISTER", exp_id, design)

def recordresult(self, expid: str, sample_id: str, result: Dict[str, Any]):
"""Record experiment result."""
Ledger.record("EXPERIMENTRESULT", uid("er-"), {
"expid": expid,
"sampleid": sample_id,
"result": result
})
```

#
==========
SECTION 23: CROSS-SCALE MAPPER
#
==========

class CrossScaleMapper:
"""Map between scales."""

def microtomacro(self, micro_vec: np.ndarray) ->np.ndarray:
"""Map micro scale vector to macro scale."""
L = max(1, int(len(micro_vec) / 2))
return micro_vec[:L]

def macrotomicro(self, macro_vec: np.ndarray) ->np.ndarray:
"""Map macro scale vector to micro scale."""
pad = np.zeros(cfg.vector_dim - len(macro_vec))
return np.concatenate([macro_vec, pad], axis=0)

#
==========
SECTION 24: MODEL REGISTRY
#
==========

class ModelRegistry:
"""Registry of trained models."""

def __init__(self):
self.lock = threading. RLock()
self.models: Dict[str, Dict[str, Any]] = {}

def register(self, name: str, metadata: Dict[str, Any]):
"""Register a model."""
with self.lock:
version = metadata.get("version", uid("v-"))
metadata["version"] = version
metadata["ts"] = now_ts()
self.models[name] = metadata

```
Ledger.record("MODEL_REGISTER", uid("mr-"), {
"name": name,
"version": version,
"meta": metadata
})

def get(self, name: str) ->Optional[Dict[str, Any]]:
"""Get model metadata."""
return self.models.get(name)

model_registry = ModelRegistry()

#
==========
# SECTION 25: DARK MATTER DETECTOR
#
==========

class DarkMatterDetector:
"""Detects anomalous dark matter signatures."""

def __init__(self, dim: int = cfg.vector_dim, usevae: bool = cfg.enablevaedark):
self.dim = dim
self.pca = PCA(n_components=min(64, dim)) if sklearn_available else None
self.ica = FastICA(n_components=min(32, dim)) if sklearn_available else None
self.iforest = IsolationForest(contamination=cfg.darkisolationcontamination) if
sklearn_available else None
self.cov = EmpiricalCovariance() if sklearn_available else None
self.warmup_data: List[np.ndarray] = []
self. _trained = False
self.usevae = usevae and torch_available
self.vae = None
Self.vaelatentdim = cfg.vanandime
self. _lock = threading. RLock()

def warmup(self, vectors: List[np.ndarray]):
"""Warmup detector with normal vectors."""
with self. _lock:
if not vectors:
return

X = np.stack(vectors, axis=0)
```

if self.pca is not None:
try:
self.pca.fit(X)
except Exception:
pass

if self.ica is not None:
try:
self.ica.fit(X)
except Exception:
pass

if self.iforest is not None:
try:
self.iforest.fit(X)
except Exception:
pass

if self.cov is not None:
try:
self.cov.fit(X)
except Exception:
pass

self.warmup_data = vectors[-2000:]
self. _trained = True
Ledger.record("DARK_WARMUP", uid("dw-"), {"samples": X.shape[0]})

def detect(self, windowvecs: List[np.ndarray],
materialvecs: Optional[List[np.ndarray]] = None) ->Dict[str, Any]:
"""Detect anomalies in vector window."""
out = {
"score": 0.0,
"zscore": 0.0,
"icacomponents": [],
"isanomaly": False,
"residual": None,
"materialexplainfraction": 0.0,
"topmaterialbases": []
}

if not windowvecs or len(windowvecs)<2:
return out

```
arr = np.stack(windowvecs, axis=0)
last = arr[-1]
mean = np.mean(arr[:-1], axis=0)
recon = mean.copy()

if self.pca is not None and self. _trained:
try:
proj = self.pca.transform(mean.reshape(1, -1))
recon = self.pca.inverse_transform(proj).reshape(-1)
except Exception:
recon = mean

residual = last - recon
out["residual"] = residual.tolist()

if self.cov is not None and self. _trained:
try:
mahal = self.cov.mahalanobis(residual.reshape(1, -1))[0]
out["zscore"] = float(mahal)
except Exception:
if self.warmup_data:
out["zscore"] = float(np.linalg.norm(residual) / (np.std(self.warmup_data) + 1e-12))
else:
out["zscore"] = float(np.linalg.norm(residual))
else:
if self.warmup_data:
out["zscore"] = float(np.linalg.norm(residual) / (np.std(self.warmup_data) + 1e-12))
else:
out["zscore"] = float(np.linalg.norm(residual))

if self.ica is not None and self. _trained:
try:
comps = self.ica.transform(residual.reshape(1, -1))[0]
out["icacomponents"] = comps.tolist()
except Exception:
out["icacomponents"] = []

isanom = False
if self.iforest is not None and self. _trained:
try:
score = self.iforest.decision_function(residual.reshape(1, -1))[0]
isanom = score<0.0
out["score"] = float(score)
```

```
except Exception:
out["score"] = -out["zscore"]
else:
out["score"] = -out["zscore"]
isanom = out["zscore"]>= cfg.darkresidualthreshold

out["isanomaly"] = bool(isanom)

if materialvecs:
try:
M = np.stack(materialvecs, axis=0)
mat_mean = np.mean(M, axis=0)
projcoeff = np.dot(residual, mat_mean) / (np.dot(mat_mean, mat_mean) + 1e-12)
out["materialexplainfraction"] = float(min(1.0, max(0.0, abs(projcoeff))))
corrs = np.dot(M, residual)
top_idx = np.argsort(-np.abs(corrs))[: min(3, M.shape[0])]
out["topmaterialbases"] = [int(i) for i in top_idx.tolist()]
except Exception:
out["materialexplainfraction"] = 0.0
out["topmaterialbases"] = []

return out

#
==========
# SECTION 26: PARALLEL ARCHITECTURE - MAIN SYSTEM
#
==========

class DeathScannerV10UltimateParallel:
"""Main parallel processing system."""

def __init__(self, sensor: Sensor, workercount: int = 4):
# Initialize all components
self.sensor = sensor
self.vectorizer = Vectorizer(dim=cfg.vector_dim)
self.materialvectorizer = MaterialVectorizer(embeddim=cfg.materialembeddim)
self.materialdecomposer = MaterialDecomposer(nbases=cfg.materialbasisk)
self.materialstability = MaterialBasisStability(self.materialdecomposer)
self.index = VectorIndex(dim=cfg.vector_dim)
self.registry = EntRegistry(path=cfg.entspath)
self.seedindex = SeedIndex(path=cfg.seedspath)
```

```
self.compressor = Compressor(self.registry, self.seedindex)
self.fusion = FusionEngine()
self.darkdetector = DarkMatterDetector(dim=cfg.vector_dim,
usevae=cfg.enablevaedark)
self.trigger = Trigger(confthreshold=cfg.trigger_confidence)
self.safety = SafetyEngine(protectedspecies=cfg.protectedspecies)
self.edgesummary = EdgeSummary(self.vectorizer, self.darkdetector)
self.experimentrunner = ExperimentRunner()
self.noisesim = NoiseSimulator()
self.synthetic = SyntheticDataGenerator(self.vectorizer, self.materialvectorizer,
self.noisesim)
self.mvf = MultiViewFactorModel(shareddim=cfg.factorshareddim)
self.monitor = monitor
self.modelregistry = model_registry

self.running = False
self.recentsamples: List[Dict[str, Any]] = []
self.materialbuffer: List[np.ndarray] = []
self.lock = threading. RLock()
self.processed = 0

# Parallel components
self.ingestthread = None
self.workerthreads: List[threading.Thread] = []
self.workercount = max(1, workercount)
self.taskqueue: "queue.Queue[Dict[str, Any]]" = queue. Queue(maxsize=4096)
self.stopevent = threading. Event()

# Background trainers
self.materialtrainerthread = None
self.vaetrainerthread = None

# ----------------------------------------------------------
# INGESTION THREAD
# ----------------------------------------------------------

def run(self):
"""Main ingestion loop - reads from sensor and enqueues."""
log.info("Ingest thread started")
while not self.stopevent.is_set():
try:
sample = self.sensor.read()
if not isinstance(sample, dict):
log.error("Sensor.read() must return dict")
```

```
time.sleep(cfg.poll_interval)
continue

# Ensure required fields
if "ts" not in sample:
sample["ts"] = time.time()
if "id" not in sample:
sample["id"] = uid("s-")
if "meta" not in sample:
sample["meta"] = {
"deviceid": "unknown",
"calibversion": "v0",
"permission_id": None,
"gps": {"lat": 0.0, "lon": 0.0}
}

# Enqueue
try:
self.taskqueue.put(sample, timeout=1.0)
except queue. Full:
log.warning(f"Task queue full; dropping sample {sample.get('id')}")

# Update recent buffer
with self.lock:
self.recentsamples.append(sample)
cutoff = time.time() - cfg.window_seconds * 2
self.recentsamples = [s for s in self.recentsamples if s["ts"]>= cutoff]

time.sleep(cfg.poll_interval)
except Exception as e:
log.error(f"Ingest loop error: {e}")
log.info("Ingest thread stopped")

# ---------------------------------------------------------
# WORKER THREADS
# ---------------------------------------------------------
def workerloop(self, wid: int):
"""Worker processing loop."""
log.info(f"Worker {wid} started")
while not self.stopevent.is_set():
try:
sample = self.taskqueue.get(timeout=1.0)
except queue. Empty:
```

```
continue

try:
self.processsample(sample)
except Exception as e:
log.error(f"Worker {wid} process error: {e}")
finally:
self.taskqueue.task_done()
log.info(f"Worker {wid} stopped")

def processsample(self, sample: Dict[str, Any]):
"""Process a single sample."""
# Schema validation
ok, reason = DataSchema.validate(sample)
if not ok:
Ledger.record("SCHEMA_VIOLATION", sample.get("id"), {"reason": reason})
return

# Permission check
okp, pr = permission_manager.check(sample.get("meta", {}).get("permissionid"))
if not okp:
Ledger.record("PERMISSION_DENIED", sample.get("id"), {"reason": pr})
return

# Safety check
ok_s, sreason = self.safety.check(sample)
if not ok_s:
Ledger.record("SAFETY_BLOCK", sample.get("id"), {"reason": sreason})
return

# Build window
with self.lock:
window = [s for s in self.recentsamples
if sample["ts"] - cfg.window_seconds<= s["ts"]<= sample["ts"]]
window = sorted(window, key=lambda x: x["ts"])

# Vectorize
vec = self.vectorizer.transform(sample, window=window)
mvec = self.materialvectorizer.transform(sample)

# Add to index and registry
self.index.add(sample["id"], vec, sample)
ent = EntNode(id=uid("ent-"), vec=vec.tolist(), shards=[sample["id"]],
score=1.0, ts=sample["ts"])
```

```
self.registry.register(ent)

# Update material buffer
self.materialbuffer.append(mvec)

# Build window vectors and material vectors
recent = self.index.queryrecent(sample["ts"], windowseconds=cfg.window_seconds,
topk=128)
windowvecs = []
target = []
materialvecs = []

for r in recent:
meta = r.get("meta", {})
metas.append({"meta": meta.get("meta", {}), "ts": meta.get("ts")})

# Find vector
found_vec = None
for n in self.registry.nodes.values():
if n.shards and r.get("id") in n.shards:
found_vec = np.array(n.vec, dtype=float)
break
if found_vec is None:
found_vec = vec
windowvecs.append(found_vec)

# Find material vector
mvecr = None
m = meta.get("meta", {})
if m and m.get("material"):
try:
mvecr = self.materialvectorizer.transform({"material": m.get("material")})
except Exception:
mvecr = None
if mvecr is None:
mvecr = mvec
materialvecs.append(mvecr)

# Fusion and dark detection
fusedscore, topfeatures = self.fusion.scorewindow(windowvecs, metas)
darkinfo = self.darkdetector.detect(windowvecs, materialvecs)

# Monitor and drift
self.monitor.record(np.array([vec.mean()]))
```

```
drifted, z = self.monitor.detectdrift()
if drifted:
log.warning(f"Data drift detected z={z:.3f}")

# Ledger and trigger
Ledger.record("SCORE", sample.get("id"), {
"score": fusedscore,
"features": topfeatures,
"dark": {
"z": darkinfo.get("zscore"),
"isanom": darkinfo.get("isanomaly"),
"materialfraction": darkinfo.get("materialexplainfraction")
}
})

action, params = self.trigger.evaluate(fusedscore, darkinfo, {"meta": sample.get("meta", {})})

if action == "dark_report":
dark_id = uid("dark-")
Ledger.record("DARKCANDIDATE", dark_id, {
"source": sample.get("id"),
"zscore": darkinfo.get("zscore"),
"score": darkinfo.get("score")
})
log.warning(f"Dark candidate {dark_id} recorded for sample {sample.get('id')} z={darkinfo.get('zscore'):.3f}")
elif action == "vocs_sample":
Ledger.record("TRIGGER_VOCS", sample.get("id"), params)
elif action == "report":
Ledger.record("TRIGGER_REPORT", sample.get("id"), params)

# Edge summary
summary = self.edgesummary.summarize(sample, window)
if summary["dist"]>0.5 or summary["darkz"]>cfg.darkresidualthreshold:
Ledger.record("EDGEBACKHAULSCHEDULED", sample.get("id"), {
"dist": summary["dist"],
"darkz": summary["darkz"]
})

# Periodic compression
try:
if len(self.registry.nodes)>500 and (len(self.registry.nodes) % 200 == 0):
res = self.compressor.buildseeds(simthresh=cfg.mergesimthreshold,
```

```
mingroup=cfg.seedmingroup)
log.info(f"Compressor build_seeds result: {res}")
except Exception as e:
log.error(f"Compressor error: {e}")

# Save replay
self.savereplay(sample)

self.processed += 1

def savereplay(self, sample: Dict[str, Any]):
"""Save sample to replay directory."""
try:
ts = sample.get("ts", time.time())
hour = int(ts // 3600)
hour_file = os.path.join(cfg.replay_dir, f"replay_{hour}.jsonl")

line = json.dumps(sample, ensure_ascii=False)
with open(hour_file, "a", encoding="utf-8") as f:
f.write(line + "\n")
except Exception as e:
log.error(f"Failed to save replay: {e}")

# ---------------------------------------------------------
# MATERIAL TRAINER THREAD
# ---------------------------------------------------------

def materialtrainer_loop(self):
"""Train material basis periodically."""
log.info("Material trainer thread started")
while not self.stopevent.is_set():
try:
if len(self.materialbuffer)>= cfg.materialbasisbootstrap:
try:
M = np.stack(self.materialbuffer[-8192:], axis=0) if len(self.materialbuffer)>= 8192 else
np.stack(self.materialbuffer, axis=0)
self.materialdecomposer.fitbasis(M, bootstrap=True)

# Bootstrap stability
def _bootstrap():
res = self.materialstability.bootstrapstability(M, niter=cfg.materialbootstrapiters)
log.info(f"Material basis stability: {res}")

threading. Thread(target=_bootstrap, daemon=True).start()
```

except Exception as e:
log.error(f"Material basis training failed: {e}")

time.sleep(5.0)
except Exception as e:
log.error(f"Material trainer loop error: {e}")
log.info("Material trainer thread stopped")

--
VAE TRAINER THREAD
--

def vaetrainer_loop(self):
"""Train VAE periodically."""
log.info("VAE trainer thread started")
while not self.stopevent.is_set():
try:
if len(self.recentsamples)>= 256:
mats = []
with self.lock:
for s in self.recentsamples[-1024:]:
try:
v = self.vectorizer.transform(s, window=[])
mats.append(v)
except Exception:
continue

if mats:
X = np.stack(mats, axis=0)

if torch_available:
model = ExplainableVAE(inputdim=X.shape[1], latentdim=cfg.vaelatentdim)
savepath = os.path.join(cfg.model_dir, "explainable_vae.pt")
try:
trainvaetorch(model, X, epochs=min(cfg.vaetrainepochs, 50),
batchsize=cfg.vaebatchsize, savepath=savepath)
self.modelregistry.register("explainablevae", {
"path": savepath,
"inputdim": X.shape[1]
})
except Exception as e:
log.error(f"VAE training failed: {e}")
else:
model = ExplainableVAE(inputdim=X.shape[1], latentdim=cfg.vaelatentdim)

try:
model.fit(X)
self.modelregistry.register("explainablevaefallback", {
"method": "svd",
"inputdim": X.shape[1]
})
except Exception as e:
log.error(f"Fallback VAE training failed: {e}")

time.sleep(30.0)
except Exception as e:
log.error(f"VAE trainer loop error: {e}")
log.info("VAE trainer thread stopped")

--
START/STOP
--

def start(self):
"""Start the system."""
self.stopevent.clear()

# Start ingest thread
self.ingestthread = threading. Thread(target=self.ingestloop, daemon=True)
self.ingestthread.start()

# Start worker threads
for i in range(self.workercount):
t = threading. Thread(target=self.workerloop, args=(i,), daemon=True)
t.start()
self.workerthreads.append(t)

# Start material trainer
self.materialtrainerthread = threading. Thread(target=self.materialtrainer_loop, daemon=True)
self.materialtrainerthread.start()

# Start VAE trainer
self.vaetrainerthread = threading. Thread(target=self.vaetrainer_loop, daemon=True)
self.vaetrainerthread.start()

log.info(f"DeathScannerV10UltimateParallel started with {self.workercount} workers")

def stop(self):

"""Stop the system."""
self.stopevent.set()

# Wait for queue to drain
try:
self.taskqueue.join(timeout=30.0)
except Exception:
pass

# Brief wait for threads
time.sleep(0.5)

# Save data
Ledger.dump()
self.registry.save()
self.seedindex.save()

log.info(f"DeathScannerV10UltimateParallel stopped. Processed {self.processed} samples")

#
==========
SECTION 27: CLI AND MAIN ENTRY POINT
#
==========

def parse_args() ->argparse. Namespace:
"""Parse command line arguments."""
parser = argparse. ArgumentParser(
description="DeathScanner V10 Ultimate - Industrial-grade parallel scanning system"
)
parser.add_argument(
"--sensor-driver",
type=str,
default="",
help="module:ClassName implementing Sensor (optional). If omitted, a SyntheticSensor will be used."
)
parser.add_argument(
"--run-seconds",
type=int,
default=0,

```
help="Run for specified seconds. 0 or omitted means run until interrupted."
)
parser.add_argument(
"--workers",
type=int,
default=4,
help="Number of worker threads for processing."
)
parser.add_argument(
"--vector-dim",
type=int,
default=256,
help="Dimension of feature vectors."
)
parser.add_argument(
"--no-permission",
action="store_true",
help="Disable permission requirement for testing."
)
parser.add_argument(
"--selftest",
action="store_true",
help="Run self-test and exit."
)
return parser.parse_args()

def loadsensorfromspecorfallback(spec: str) ->Sensor:
"""Load sensor from spec or use fallback."""
if not spec:
log.info("No sensor-driver specified; using SyntheticSensor fallback")
return SyntheticSensor()

try:
modulename, classname = spec.split(":")
module = __import__(modulename, fromlist=[classname])
cls = getattr(module, classname)
inst = cls()
if not isinstance(inst, Sensor):
raise RuntimeError("Loaded class is not a Sensor subclass")
return inst
except Exception as e:
log.error(f"Failed to load sensor driver '{spec}': {e}; falling back to SyntheticSensor")
return SyntheticSensor()
```

```
def run_selftest():
"""Run basic self-test."""
log.info("=== Running Self-Test ===")

# Test vectorizer
sample = {
"ts": time.time(),
"id": "test-001",
"meta": {"deviceid": "test", "calibversion": "v1", "permissionid": "test", "gps": {"lat": 0,
"lon": 0}},
"phys": {"chlf": 1.0, "temp": 25.0, "acoustic": 0.5, "microelectrode": 0.1, "vocs": [0.1]*8,
"mass_spec": [0.2]*16},
"material": {"mass_spec": [0.1]*64, "raman": [0.1]*128, "swir": [0.1]*64}
}

try:
vectorizer = Vectorizer()
vectorizer.fit_warmup([sample])
vec = vectorizer.transform(sample)
log.info(f"✓ Vectorizer: dim={len(vec)}, norm={np.linalg.norm(vec):.4f}")
except Exception as e:
log.error(f"✗ Vectorizer failed: {e}")

# Test material vectorizer
try:
mv = MaterialVectorizer()
mv.fit_warmup([sample])
mvec = mv.transform(sample)
log.info(f"✓ MaterialVectorizer: dim={len(mvec)}, norm={np.linalg.norm(mvec):.4f}")
except Exception as e:
log.error(f"✗ MaterialVectorizer failed: {e}")

# Test dark detector
try:
detector = DarkMatterDetector()
detector.warmup([vec, vec + 0.1])
result = detector.detect([vec, vec + 0.5])
log.info(f"✓ DarkMatterDetector: score={result['score']:.4f},
isanomaly={result['isanomaly']}")
except Exception as e:
log.error(f"✗ DarkMatterDetector failed: {e}")

# Test VAE
try:
```

```
X = np.random.randn(100, cfg.vector_dim)
vae = ExplainableVAE(inputdim=cfg.vector_dim, latentdim=16)
vae.fit(X)
z = vae.encode(X[:1])
log.info(f"✓ ExplainableVAE: latent_dim={len(z[0])}")
except Exception as e:
log.error(f"✗ ExplainableVAE failed: {e}")

# Test fusion engine
try:
fusion = FusionEngine()
score, feats = fusion.scorewindow([vec, vec + 0.1], [{"meta": {"temp": 25}}, {"meta": {"temp": 26}}])
log.info(f"✓ FusionEngine: score={score:.4f}, features={feats}")
except Exception as e:
log.error(f"✗ FusionEngine failed: {e}")

log.info("=== Self-Test Complete ===")

def main():
"""Main entry point."""
args = parse_args()

# Apply config overrides
if args.no_permission:
cfg.require_permission = False
if args.vector_dim:
cfg.vector_dim = args.vector_dim

# Run self-test if requested
if args.selftest:
run_selftest()
return

# Load sensor
sensor = loadsensorfromspecorfallback(args.sensor_driver)

# Create and start system
ds = DeathScannerV10UltimateParallel(sensor, workercount=args.workers)
ds.start()

# Run for specified time or indefinitely
run_seconds = args.run_seconds if args.run_seconds>0 else None
```

```
def signal_handler(sig, frame):
log.info("Received interrupt signal, shutting down...")
ds.stop()
sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)

try:
if run_seconds:
log.info(f"Running for {run_seconds} seconds...")
time.sleep(run_seconds)
else:
log.info("Running until interrupted (Ctrl+C)...")
while True:
time.sleep(1.0)
except KeyboardInterrupt:
log.info("Interrupted by user")
except Exception as e:
log.error(f"Unexpected error: {e}")
finally:
ds.stop()

if __name__ == "__main__":
main()
```