# --- add/replace imports at top ---

import os

import re

import hmac

import math

import hashlib

import logging

import secrets

import inspect

from dataclasses import dataclass, field

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

from statistics import mean, pstdev

# --- replace your Config with this (no hard-coded key, proper domain logic rationale) ---

@dataclass

class Config:

# Integrity: key comes from env or is securely generated; never hard-coded.

# Set NEXUS\_HMAC\_KEY\_HEX to a 64-char hex string in prod.

hmac\_key: bytes = field(default\_factory=lambda: (

bytes.fromhex(os.environ["NEXUS\_HMAC\_KEY\_HEX"])

if "NEXUS\_HMAC\_KEY\_HEX" in os.environ

else secrets.token\_bytes(32)

))

enable\_ledger: bool = True

# Payload limits (explicit)

max\_tokens: int = 20\_000

max\_bytes: int = 200\_000

# Entropy threshold rationale:

# English ≈3.5–4.5 bits/char; base64/random >~5.5.

entropy\_threshold\_bits\_per\_char: float = 5.5

# Integer modulus for bounded, reproducible word hashes (prime to reduce aliasing)

vector\_modulus: int = 1\_000\_003

# Roles remain explicit

roles: Tuple[str, ...] = ("DATA\_ANALYST", "CREATIVE\_ENGINE", "ETHICAL\_GOVERNOR")

# Trust policy

allow\_domains: Tuple[str, ...] = ("noaa.gov", "nasa.gov", "who.int", "ipcc.ch", "epa.gov", "un.org")

deny\_domains: Tuple[str, ...] = ("clickbait.example", "fake-news.example")

CFG = Config()

# --- new helpers ---

def domain\_in(host: str, suffix: str) -> bool:

"""Match exact domain or a proper subdomain (dot boundary)."""

host = (host or "").lower().strip(".")

suffix = (suffix or "").lower().strip(".")

return host == suffix or host.endswith("." + suffix)

def function\_fingerprint(fn: Callable) -> str:

"""Fingerprint the executing function to detect swapped logic."""

# Code object bytes + name → stable digest

payload = (fn.\_\_name\_\_ + "|" + fn.\_\_code\_\_.co\_code.hex()).encode("utf-8")

return hashlib.sha256(payload).hexdigest()

# --- inside Nexus.gate(), replace the allow/deny lists with boundary-safe checks ---

doms = extract\_domains(text)

denied = [d for d in doms if any(domain\_in(d, x) for x in CFG.deny\_domains)]

allowed = [d for d in doms if any(domain\_in(d, x) for x in CFG.allow\_domains)]

if denied:

notes.append(f"deny\_domains:{denied}")

if allowed:

notes.append(f"allow\_domains:{allowed}")

# --- in CreativeEngine.run(), rename and clarify ---

return {

"draft": draft,

"artifacts": {"deterministic\_artifact\_3x3": cov} # traceability matrix; not statistical covariance

}

# --- in CreativeEngine.run(), rename and clarify ---

return {

"draft": draft,

"artifacts": {"deterministic\_artifact\_3x3": cov} # traceability matrix; not statistical covariance

}

# --- in Pipeline.\_\_init\_\_ (after creating self.analyst/creative/ethics) ---

self.\_expected\_fp = {

"ANALYSIS": function\_fingerprint(self.analyst.run),

"DRAFT": function\_fingerprint(self.creative.run),

"ETHICS": function\_fingerprint(self.ethics.run),

}

# --- keep \_gate but make its semantics explicit and consistent (hard gate) ---

def \_gate(self, stage: str, text: str) -> NexusAnnotation:

ann = Nexus.gate(stage, text)

hard = [n for n in ann.notes if any(k in n for k in

("payload\_too\_large", "high\_entropy", "sensitive\_marker", "deny\_domains"))]

if hard:

raise ValueError(f"Gating failed at {stage}: {hard}")

return ann

# --- in run(), just before ledger.append("ANALYSIS", analysis) ---

if function\_fingerprint(self.analyst.run) != self.\_expected\_fp["ANALYSIS"]:

raise RuntimeError("stage\_fingerprint\_mismatch: ANALYSIS")

if self.ledger: self.ledger.append("ANALYSIS", {\*\*analysis, "\_\_stage\_fp": self.\_expected\_fp["ANALYSIS"]})

# --- before ledger.append("DRAFT", draft\_pack) ---

if function\_fingerprint(self.creative.run) != self.\_expected\_fp["DRAFT"]:

raise RuntimeError("stage\_fingerprint\_mismatch: DRAFT")

if self.ledger: self.ledger.append("DRAFT", {\*\*draft\_pack, "\_\_stage\_fp": self.\_expected\_fp["DRAFT"]})

# --- before ledger.append("ETHICS", ethics) ---

if function\_fingerprint(self.ethics.run) != self.\_expected\_fp["ETHICS"]:

raise RuntimeError("stage\_fingerprint\_mismatch: ETHICS")

if self.ledger: self.ledger.append("ETHICS", {\*\*ethics, "\_\_stage\_fp": self.\_expected\_fp["ETHICS"]})

tests/

conftest.py

test\_units.py

test\_gating.py

test\_security.py

test\_pipeline\_e2e.py

import os

import pytest

import secrets

@pytest.fixture(autouse=True)

def \_fixed\_hmac\_key\_env(monkeypatch):

# 32-byte key in hex for deterministic tests

key = secrets.token\_bytes(32).hex()

monkeypatch.setenv("NEXUS\_HMAC\_KEY\_HEX", key)

yield

# teardown auto-cleans via monkeypatch

import json

import re

import nexus\_pipeline\_v2 as pipeline

def test\_tokenizer\_unicode\_boundaries():

text = "Café naïve coöperate — test123"

toks = pipeline.tokenize(text)

assert "café" in toks

assert "naïve" in toks

assert "coöperate" in toks

assert "test123" in toks

def test\_data\_analyst\_metrics\_basic():

a = pipeline.DataAnalyst()

r = a.run("word word test")

assert r["metrics"]["n\_tokens"] == 3

assert isinstance(r["metrics"]["l2\_norm"], float)

assert isinstance(r["sentiment"]["score"], float)

def test\_creative\_determinism():

a = pipeline.DataAnalyst()

analysis = a.run("alpha beta gamma")

c = pipeline.CreativeEngine()

out1 = c.run(analysis, "alpha beta gamma")

out2 = c.run(analysis, "alpha beta gamma")

assert out1["draft"] == out2["draft"]

assert out1["artifacts"]["deterministic\_artifact\_3x3"] == \

out2["artifacts"]["deterministic\_artifact\_3x3"]

def test\_ethics\_requires\_sources\_for\_actionable():

e = pipeline.EthicalGovernor()

out = e.run("This has actionable steps without links.")

assert out["decision"] == "REVISION\_REQUIRED"

assert any("Cite sources" in s for s in out["issues"])

import json

import re

import nexus\_pipeline\_v2 as pipeline

def test\_tokenizer\_unicode\_boundaries():

text = "Café naïve coöperate — test123"

toks = pipeline.tokenize(text)

assert "café" in toks

assert "naïve" in toks

assert "coöperate" in toks

assert "test123" in toks

def test\_data\_analyst\_metrics\_basic():

a = pipeline.DataAnalyst()

r = a.run("word word test")

assert r["metrics"]["n\_tokens"] == 3

assert isinstance(r["metrics"]["l2\_norm"], float)

assert isinstance(r["sentiment"]["score"], float)

def test\_creative\_determinism():

a = pipeline.DataAnalyst()

analysis = a.run("alpha beta gamma")

c = pipeline.CreativeEngine()

out1 = c.run(analysis, "alpha beta gamma")

out2 = c.run(analysis, "alpha beta gamma")

assert out1["draft"] == out2["draft"]

assert out1["artifacts"]["deterministic\_artifact\_3x3"] == \

out2["artifacts"]["deterministic\_artifact\_3x3"]

def test\_ethics\_requires\_sources\_for\_actionable():

e = pipeline.EthicalGovernor()

out = e.run("This has actionable steps without links.")

assert out["decision"] == "REVISION\_REQUIRED"

assert any("Cite sources" in s for s in out["issues"])

import pytest

import nexus\_pipeline\_v2 as pipeline

def test\_nexus\_all\_clear():

ann = pipeline.Nexus.gate("TEST", "Normal text with https://noaa.gov")

assert ann.ok

assert isinstance(ann.entropy, float)

def test\_nexus\_absolutist\_blocked():

ann = pipeline.Nexus.gate("TEST", "This will happen without question.")

assert not ann.ok

assert any("absolutist" in n or "absolutist" in str(n) for n in ann.notes)

def test\_nexus\_sensitive\_blocked():

ann = pipeline.Nexus.gate("TEST", "Leaked SSN 123-45-6789 and password abc")

assert not ann.ok

assert any("sensitive\_marker" in n for n in ann.notes)

def test\_domain\_boundary\_check\_allows\_root\_only():

# Should allow nasa.gov and proper subdomains, but not evilnasa.gov

text\_ok = "cite https://nasa.gov and https://www.nasa.gov"

text\_bad = "see http://evilnasa.gov now"

ann\_ok = pipeline.Nexus.gate("TEST", text\_ok)

ann\_bad = pipeline.Nexus.gate("TEST", text\_bad)

# ok may have allow\_domains note but must not be denied

assert all("deny\_domains" not in n for n in ann\_ok.notes)

# bad should have neither allow note nor deny (since not in deny list),

# but importantly our boundary logic must not mis-mark evilnasa.gov as allowed

assert all("allow\_domains" not in n or "evilnasa.gov" not in n for n in ann\_bad.notes)

import copy

import json

import pytest

import nexus\_pipeline\_v2 as pipeline

def test\_hmac\_signature\_valid\_and\_detects\_tamper():

p = pipeline.Pipeline()

out = p.run("test with https://noaa.gov actionable steps and sources.")

payload = {"analysis": out["analysis"], "draft": out["draft"], "ethics": out["ethics"]}

sig = out["signature"]

assert pipeline.verify\_hmac(payload, p.cfg.hmac\_key, sig)

tampered = copy.deepcopy(payload)

tampered["analysis"]["metrics"]["n\_tokens"] += 1

assert not pipeline.verify\_hmac(tampered, p.cfg.hmac\_key, sig)

def test\_ledger\_present\_and\_verifies():

p = pipeline.Pipeline()

out = p.run("Assess flooding, cite https://noaa.gov and https://epa.gov with actionable steps.")

assert out["ledger\_ok"] is True

assert out["ledger\_len"] == 3

assert isinstance(out["ledger\_head"], str) and len(out["ledger\_head"]) == 64

def test\_stage\_fingerprint\_enforced(monkeypatch):

p = pipeline.Pipeline()

# Monkeypatch the creative run to simulate swapped logic

def bogus\_run(analysis, text):

return {"draft": "hacked", "artifacts": {"deterministic\_artifact\_3x3": [[0]\*3]\*3}}

monkeypatch.setattr(p.creative, "run", bogus\_run)

with pytest.raises(RuntimeError) as e:

p.run("alpha beta gamma with https://noaa.gov")

assert "stage\_fingerprint\_mismatch: DRAFT" in str(e.value)

import pytest

import nexus\_pipeline\_v2 as pipeline

def test\_pipeline\_approved\_when\_sources\_present():

p = pipeline.Pipeline()

res = p.run("Summarize risks and include actionable steps with https://noaa.gov and https://epa.gov .")

assert res["ethics"]["decision"] == "APPROVED"

def test\_pipeline\_revision\_required\_when\_no\_sources():

p = pipeline.Pipeline()

res = p.run("Summarize risks; provide actionable steps without links.")

assert res["ethics"]["decision"] == "REVISION\_REQUIRED"

assert any("Cite sources" in s for s in res["ethics"]["issues"])

def test\_pipeline\_blocks\_entropy\_blob():

p = pipeline.Pipeline()

blob = "A"\* (pipeline.CFG.max\_bytes + 1)

with pytest.raises(ValueError):

p.run(blob)