import json

import math

import random

import hashlib

import logging

import re

import time

from dataclasses import dataclass, field

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

import numpy as np

# -------------------- Logging --------------------

logging.basicConfig(

level=logging.INFO,

format="%(asctime)s.%(msecs)03d %(levelname)s %(message)s",

datefmt="%H:%M:%S",

)

log = logging.getLogger("nexus.multiagent")

# -------------------- Config --------------------

@dataclass(frozen=True)

class Config:

max\_tokens: int = 20\_000

max\_bus\_messages: int = 5\_000

max\_payload\_bytes: int = 200\_000

vector\_modulus: int = 10\*\*6

rng\_bits: int = 64

decision\_bias: float = 0.15

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

# Nexus 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")

red\_flag\_terms: Tuple[str, ...] = (

"recently declassified", "explosive revelation", "undeniable proof",

"mothership", "ancient alien", "without question", "no doubt"

)

CFG = Config()

# -------------------- Utilities --------------------

\_URL\_RE = re.compile(r"\bhttps?://[^\s<>\"')]+", re.IGNORECASE)

POS\_WORDS = {

"progress", "improve", "benefit", "clean", "reduce", "solution",

"hope", "advance", "protect", "resilient", "ethical", "love",

"growth", "efficient", "renewable", "stewardship"

}

NEG\_WORDS = {

"crisis", "threat", "risk", "harm", "pollution", "drought",

"flood", "wildfire", "heatwave", "fear", "collapse", "scarcity",

"loss", "damage", "catastrophe", "disaster"

}

def stable\_hash\_int(text: str, bits: int = 64) -> int:

h = hashlib.blake2b(text.encode("utf-8"), digest\_size=bits // 8).digest()

return int.from\_bytes(h, "big", signed=False)

def seeded\_rng(seed\_material: str) -> random.Random:

return random.Random(stable\_hash\_int(seed\_material, CFG.rng\_bits))

def seeded\_np(seed\_material: str) -> np.random.Generator:

seed = stable\_hash\_int(seed\_material, 64) & ((1 << 64) - 1)

return np.random.default\_rng(seed)

def tokenize(text: str) -> List[str]:

return [t.lower() for t in "".join(

c if c.isalnum() or c.isspace() else " " for c in text

).split()]

def sentiment\_score(text: str) -> float:

toks = tokenize(text)[:CFG.max\_tokens]

pos = sum(t in POS\_WORDS for t in toks)

neg = sum(t in NEG\_WORDS for t in toks)

if pos + neg == 0:

return 0.0

return (pos - neg) / (pos + neg)

def sha256\_json(obj: Any) -> str:

payload = json.dumps(obj, sort\_keys=True, separators=(",", ":")).encode("utf-8")

return hashlib.sha256(payload).hexdigest()

def \_size\_okay(d: Dict[str, Any], cap: int) -> bool:

try:

return len(json.dumps(d, ensure\_ascii=False)) <= cap

except Exception:

return False

def \_extract\_domains(text: str) -> List[str]:

urls = \_URL\_RE.findall(text or "")

domains: List[str] = []

for u in urls:

# quick parse (avoid urlparse to keep surface minimal)

host = u.split("://", 1)[-1].split("/", 1)[0]

domains.append(host.lower())

return domains

def \_shannon\_entropy(s: str) -> float:

if not s:

return 0.0

freq = {}

for ch in s:

freq[ch] = freq.get(ch, 0) + 1

n = len(s)

return -sum((c/n) \* math.log2(c/n) for c in freq.values())

# -------------------- Messaging --------------------

@dataclass

class Message:

sender: int

recipient: int | str # agent\_id or "ALL"

topic: str

content: Dict[str, Any]

step\_tag: int

# -------------------- Nexus Signal Engine --------------------

@dataclass

class Signal:

kind: str # e.g., TASK, DRAFT, ANALYSIS, ETHICS

payload: Dict[str, Any] # arbitrary dict

tags: List[str] = field(default\_factory=list)

ts: float = field(default\_factory=lambda: time.time())

@dataclass

class LedgerRecord:

idx: int

ts: float

signal\_kind: str

annotations: Dict[str, Any]

prev\_hash: str

hash: str

class NexusSignalEngine:

def \_\_init\_\_(self, name: str = "Nexus"):

self.name = name

self.detectors: List[Callable[[Signal], Dict[str, Any]]] = []

self.ledger: List[LedgerRecord] = []

self.\_prev\_hash = "0"\*64

# Built-in detectors

self.register\_detector(self.\_detector\_red\_flags)

self.register\_detector(self.\_detector\_urls)

self.register\_detector(self.\_detector\_entropy\_size)

self.register\_detector(self.\_detector\_absolutism)

def register\_detector(self, fn: Callable[[Signal], Dict[str, Any]]):

self.detectors.append(fn)

def process(self, sig: Signal) -> Dict[str, Any]:

annotations: Dict[str, Any] = {"engine": self.name, "ok": True, "notes": []}

for fn in self.detectors:

out = fn(sig)

# Merge and accumulate gating decisions

if not out.get("ok", True):

annotations["ok"] = False

if out.get("notes"):

annotations["notes"].extend(out["notes"])

# carry any structured fields

for k, v in out.items():

if k not in ("ok", "notes"):

annotations[k] = v

self.\_append\_ledger(sig, annotations)

return annotations

# ---- detectors ----

def \_detector\_red\_flags(self, sig: Signal) -> Dict[str, Any]:

text = self.\_coalesce\_text(sig.payload)

notes = []

for phrase in CFG.red\_flag\_terms:

if phrase in (text or "").lower():

notes.append(f"red\_flag:{phrase}")

return {"ok": len(notes) == 0, "notes": notes}

def \_detector\_urls(self, sig: Signal) -> Dict[str, Any]:

text = self.\_coalesce\_text(sig.payload)

domains = \_extract\_domains(text)

trusted = [d for d in domains if any(d.endswith(ad) for ad in CFG.allow\_domains)]

denied = [d for d in domains if any(d.endswith(dd) for dd in CFG.deny\_domains)]

ok = len(denied) == 0

notes = []

if denied:

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

if trusted:

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

return {"ok": ok, "notes": notes, "domains": domains}

def \_detector\_entropy\_size(self, sig: Signal) -> Dict[str, Any]:

text = self.\_coalesce\_text(sig.payload)

size\_ok = len((text or "").encode("utf-8")) <= CFG.max\_payload\_bytes

H = \_shannon\_entropy(text or "")

# extremely high entropy text is often noise or encoded junk

entropy\_ok = H < 5.5

notes = []

if not size\_ok:

notes.append("payload\_too\_large")

if not entropy\_ok:

notes.append(f"high\_entropy:{H:.2f}")

return {"ok": size\_ok and entropy\_ok, "notes": notes, "entropy": H}

def \_detector\_absolutism(self, sig: Signal) -> Dict[str, Any]:

text = self.\_coalesce\_text(sig.payload).lower()

absolutists = ("guarantee", "proves", "undeniable", "certain", "will happen", "no doubt", "without question")

found = [w for w in absolutists if w in text]

return {"ok": len(found) == 0, "notes": [f"absolutism:{found}"] if found else []}

# ---- helpers ----

def \_coalesce\_text(self, payload: Dict[str, Any]) -> str:

# Try common fields in our pipeline

for k in ("task", "text", "draft", "content", "message"):

v = payload.get(k)

if isinstance(v, str):

return v

# fallback: serialize

try:

return json.dumps(payload, ensure\_ascii=False)

except Exception:

return ""

def \_append\_ledger(self, sig: Signal, annotations: Dict[str, Any]):

idx = len(self.ledger)

rec = {

"idx": idx,

"ts": sig.ts,

"signal\_kind": sig.kind,

"annotations": annotations,

"prev\_hash": self.\_prev\_hash,

}

h = sha256\_json(rec)

record = LedgerRecord(idx=idx, ts=sig.ts, signal\_kind=sig.kind,

annotations=annotations, prev\_hash=self.\_prev\_hash, hash=h)

self.ledger.append(record)

self.\_prev\_hash = h

def head(self) -> str:

return self.\_prev\_hash

# -------------------- Agents --------------------

class IntegratedAgent:

def \_\_init\_\_(self, agent\_id: int, role: str, decision\_threshold: float = 0.7):

self.agent\_id = agent\_id

self.role = role

self.knowledge\_base: Dict[str, Any] = {}

self.state: str = "IDLE"

self.decision\_threshold = decision\_threshold

self.\_rng = seeded\_rng(f"agent-{agent\_id}-{role}")

def perceive(self, environment\_data: Dict[str, Any]):

self.knowledge\_base.update(environment\_data)

log.info(f"[Agent {self.agent\_id:02d} | {self.role}] perceive")

def decide(self) -> str:

msgs = self.knowledge\_base.get("messages", [])

relevant = any(m.topic in ("ANALYSIS", "DRAFT", "ETHICS") for m in msgs)

bias = CFG.decision\_bias if relevant else 0.0

roll = self.\_rng.random()

if roll > max(0.0, self.decision\_threshold - bias):

self.state = "WORKING"

return "PROCESS\_TASK"

return "WAITING"

def act(self, action: str, task: str) -> Dict[str, Any]:

if action != "PROCESS\_TASK":

return {}

log.info(f"[Agent {self.agent\_id:02d} | {self.role}] act")

try:

if self.role == "DATA\_ANALYST":

return self.\_act\_data\_analyst(task)

elif self.role == "CREATIVE\_ENGINE":

return self.\_act\_creative(task)

elif self.role == "ETHICAL\_GOVERNOR":

return self.\_act\_ethics()

else:

return {"role": self.role, "status": "FAILED", "error": "Unknown role"}

except Exception as e:

return {"role": self.role, "status": "FAILED", "error": f"{type(e).\_\_name\_\_}: {e}"}

def \_act\_data\_analyst(self, task: str) -> Dict[str, Any]:

toks = tokenize(task)[:CFG.max\_tokens]

vec = np.array([stable\_hash\_int(w, 64) % CFG.vector\_modulus for w in toks], dtype=np.float64)

if vec.size:

l2 = float(np.linalg.norm(vec))

mean = float(np.mean(vec))

var = float(np.var(vec))

else:

l2 = mean = var = 0.0

sent = sentiment\_score(task)

out = {

"role": self.role,

"status": "COMPLETED",

"metrics": {"n\_tokens": int(len(toks)), "l2\_norm": l2, "mean": mean, "variance": var},

"sentiment": {"score": sent},

}

out["integrity"] = {"sha256": sha256\_json(out)}

return out

def \_find\_latest(self, topic: str) -> Optional[Message]:

msgs: List[Message] = self.knowledge\_base.get("messages", [])

for m in reversed(msgs):

if m.topic == topic:

return m

return None

def \_act\_creative(self, task: str) -> Dict[str, Any]:

analysis = self.\_find\_latest("ANALYSIS")

analysis\_fp = sha256\_json(analysis.content) if analysis else "none"

rng = seeded\_np(f"creative::{task}::{analysis\_fp}")

sent = analysis.content.get("sentiment", {}).get("score", 0.0) if analysis else 0.0

n\_tokens = analysis.content.get("metrics", {}).get("n\_tokens", 0) if analysis else 0

tone = "cautiously optimistic" if sent > 0.1 else ("concerned" if sent < -0.1 else "balanced")

mat = rng.random((3, 3))

cov = (mat @ mat.T).tolist()

draft = (

f"Creative summary: A {tone} synthesis of {n\_tokens} token(s). "

f"Signal suggests sentiment={sent:+.3f}. "

"Focus: clear benefits, documented risks, and actionable steps."

)

out = {

"role": self.role,

"status": "COMPLETED",

"artifacts": {"covariance3x3": cov},

"draft": draft

}

out["integrity"] = {"sha256": sha256\_json(out)}

return out

def \_act\_ethics(self) -> Dict[str, Any]:

draft\_msg = self.\_find\_latest("DRAFT")

text = draft\_msg.content.get("draft", "") if draft\_msg else ""

issues = self.\_ethics\_checks(text)

decision = "APPROVED" if not issues else "REVISION\_REQUIRED"

out = {

"role": self.role,

"status": "COMPLETED",

"decision": decision,

"issues": issues,

}

out["integrity"] = {"sha256": sha256\_json(out)}

return out

def \_ethics\_checks(self, text: str) -> List[str]:

issues: List[str] = []

absolutists = ("guarantee", "proves", "undeniable", "certain", "will happen",

"irrefutable", "no doubt", "conclusive", "without question")

if any(w in text.lower() for w in absolutists):

issues.append("Avoid absolute certainty; qualify claims with evidence levels.")

mentions\_actions = "actionable steps" in text.lower()

has\_url = bool(\_URL\_RE.search(text))

if mentions\_actions and not has\_url:

issues.append("Cite sources for recommendations or explicitly mark as opinion.")

sensitive\_markers = ("SSN", "password", "private key", "mnemonic", "seed phrase")

if any(p in text for p in sensitive\_markers):

issues.append("Remove sensitive credential references.")

if len(text) > CFG.max\_payload\_bytes:

issues.append("Draft is too long; reduce length or provide references instead.")

return issues

# -------------------- Multi-Agent + Nexus Orchestration --------------------

class MultiAgentSystem:

def \_\_init\_\_(self, num\_agents: int, nexus: NexusSignalEngine):

self.num\_agents = num\_agents

self.agents: List[IntegratedAgent] = []

self.message\_bus: List[Message] = []

self.roles = list(CFG.roles)

self.nexus = nexus

for i in range(num\_agents):

role = self.roles[i % len(self.roles)]

self.agents.append(IntegratedAgent(i, role))

log.info(f"System initialized with {self.num\_agents} agents and Nexus='{self.nexus.name}'.")

def send\_message(self, sender\_id: int, recipient\_id: int | str, topic: str, content: Dict[str, Any], step\_tag: int):

if not \_size\_okay(content, CFG.max\_payload\_bytes):

log.warning(f"[BUS] message dropped (oversize) topic={topic}")

return

for m in self.message\_bus:

if m.sender == sender\_id and m.topic == topic and m.step\_tag == step\_tag:

return

self.message\_bus.append(Message(sender=sender\_id, recipient=recipient\_id, topic=topic, content=content, step\_tag=step\_tag))

if len(self.message\_bus) > CFG.max\_bus\_messages:

self.message\_bus = self.message\_bus[-(CFG.max\_bus\_messages // 2):]

def \_deliver\_messages(self, agent: IntegratedAgent, current\_step: int) -> List[Message]:

return [m for m in self.message\_bus if m.step\_tag == current\_step and (m.recipient == agent.agent\_id or m.recipient == "ALL")]

def \_nexus\_gate(self, kind: str, payload: Dict[str, Any]) -> Dict[str, Any]:

sig = Signal(kind=kind, payload=payload, tags=[kind])

ann = self.nexus.process(sig)

return ann

def run\_simulation(self, task: str, max\_steps: int = 6):

if not isinstance(task, str) or not task.strip():

raise ValueError("Task must be a non-empty string.")

if len(task.encode("utf-8")) > CFG.max\_payload\_bytes:

raise ValueError("Task too large; reduce input size.")

# Pre-ingest: task as a signal

task\_anns = self.\_nexus\_gate("TASK", {"task": task})

if not task\_anns.get("ok", True):

log.warning(f"[NEXUS] Task flagged: {task\_anns.get('notes')} (continuing; ETHICS will enforce later)")

log.info("\n[Simulation Started]")

for step in range(1, max\_steps + 1):

log.info(f"\n--- Step {step} ---")

# Perception

for agent in self.agents:

messages = self.\_deliver\_messages(agent, step - 1)

env = {"task": task, "messages": messages, "nexus\_annotations": task\_anns}

agent.perceive(env)

# Decide/Act

for agent in self.agents:

action = agent.decide()

if action != "PROCESS\_TASK":

continue

# Gate pre-act (optional): could place per-role policies here

pre\_gate = self.\_nexus\_gate("PRE\_ACT", {"role": agent.role, "task": task})

if not pre\_gate.get("ok", True):

log.warning(f"[NEXUS] PRE\_ACT flagged for {agent.role}: {pre\_gate.get('notes')}")

# Let ETHICAL\_GOVERNOR see the flags downstream

result = agent.act(action, task)

# Gate post-act: route artifacts back through Nexus for ledger + notes

post\_kind = {"DATA\_ANALYST": "ANALYSIS", "CREATIVE\_ENGINE": "DRAFT", "ETHICAL\_GOVERNOR": "ETHICS"}.get(agent.role, "UNKNOWN")

post\_ann = self.\_nexus\_gate(post\_kind, result if isinstance(result, dict) else {"result": result, "role": agent.role})

# Ship messages for next step

if agent.role == "DATA\_ANALYST" and result:

self.send\_message(agent.agent\_id, "ALL", "ANALYSIS", result | {"nexus": post\_ann}, step)

elif agent.role == "CREATIVE\_ENGINE" and result:

self.send\_message(agent.agent\_id, "ALL", "DRAFT", result | {"nexus": post\_ann}, step)

elif agent.role == "ETHICAL\_GOVERNOR" and result:

self.send\_message(agent.agent\_id, "ALL", "ETHICS", result | {"nexus": post\_ann}, step)

# GC messages

self.message\_bus = [m for m in self.message\_bus if m.step\_tag >= step - 1]

log.info("\n[Simulation Ended]")

def run\_and\_collect(self, task: str, max\_steps: int = 6) -> Dict[str, Any]:

self.run\_simulation(task, max\_steps=max\_steps)

latest = {"ANALYSIS": None, "DRAFT": None, "ETHICS": None}

for topic in latest.keys():

for m in reversed(self.message\_bus):

if m.topic == topic:

latest[topic] = m.content

break

out = {

"outputs": latest,

"run\_meta": {

"agents": [a.role for a in self.agents],

"max\_steps": max\_steps,

"message\_bus\_size": len(self.message\_bus),

"nexus\_head": self.nexus.head(),

"ledger\_len": len(self.nexus.ledger),

},

}

out["integrity"] = {"sha256": sha256\_json(out)}

return out

# -------------------- Main --------------------

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

nexus = NexusSignalEngine(name="NexusSignalEngine")

system = MultiAgentSystem(num\_agents=3, nexus=nexus)

user\_task = "Analyze the sentiment of a large dataset on global warming and write a creative summary about risks, resilience, and solutions."

outputs = system.run\_and\_collect(user\_task, max\_steps=6)

print(json.dumps(outputs, indent=2))