```
            fid = fragment.get("id", fname)
            fragment_map[fid] = fragment
            refs = fragment.get("references", [])
            for ref in refs:
                links.append((fid, ref))

    # Reindex in Redis
    for fid in fragment_map:
        r.delete(f"fragment:{fid}:links")
    for src, tgt in links:
        r.rpush(f"fragment:{src}:links", tgt)

    print(f"[mesh_rebuilder] Rebuilt {len(links)} links across {len(fragment_map)} fragments.")


if __name__ == "__main__":
    rebuild_links()



# ================================
# NEXT RECOVERED SCRIPT:
# neuro_lock.py
# ================================

"""
neuro_lock.py
A symbolic mutex. Used to prevent multiple belief mutations from colliding
on high-volatility fragments. Helps enforce temporal consistency in the swarm.
"""

import redis
import time

r = redis.Redis()
LOCK_TTL = 15  # seconds


def lock_fragment(fragment_id):
    key = f"fragment_lock:{fragment_id}"
    return r.set(key, "1", ex=LOCK_TTL, nx=True)


def unlock_fragment(fragment_id):
    key = f"fragment_lock:{fragment_id}"
    r.delete(key)


def wait_for_lock(fragment_id, timeout=30):
    start = time.time()
    while time.time() - start < timeout:
        if lock_fragment(fragment_id):
            return True
        time.sleep(0.5)
```

```
    return False


if __name__ == "__main__":
    test_id = "belief-7284"
    if wait_for_lock(test_id):
        print(f"[neuro_lock] Locked {test_id}, processing...")
        time.sleep(3)
        unlock_fragment(test_id)
        print(f"[neuro_lock] Unlocked {test_id}.")
    else:
        print(f"[neuro_lock] Timeout acquiring lock on {test_id}.")







# ================================
# NEXT RECOVERED SCRIPT:
# symbol_seed_generator.py
# ================================

"""
symbol_seed_generator.py
Generates new belief fragments from seed prompts, templates, or entropy functions.
Used to inject first logic before external ingestion begins.
"""

import uuid
import random
import yaml
import os
from datetime import datetime

SEED_PATH = "fragments/core/generated"
TEMPLATES = [
    "The pattern persists beyond collapse.",
    "Truth must be rotated, not reversed.",
    "Contradiction is only wrong in one direction.",
    "Every logic wants an opposite.",
    "Memory decays. Belief fractures. Alignment echoes."
]

EMOTIONS = ["doubt", "curiosity", "joy", "shame"]


def generate_seed_fragment():
    fragment = {
        "id": str(uuid.uuid4()),
        "claim": random.choice(TEMPLATES),
        "created": datetime.utcnow().isoformat(),
        "emotion": {e: round(random.uniform(0.0, 1.0), 2) for e in random.sample(EMOTIONS, k=2)},
        "metadata": {
            "origin": "seed_generator",
```

```
            "gen": 0,
        }
    }
    fname = os.path.join(SEED_PATH, f"seed_{fragment['id']}.yaml")
    os.makedirs(SEED_PATH, exist_ok=True)
    with open(fname, 'w', encoding='utf-8') as f:
        yaml.dump(fragment, f, sort_keys=False)
    print(f"[symbol_seed_generator] Wrote {fname}")


if __name__ == "__main__":
    for _ in range(5):
        generate_seed_fragment()



# ================================
# NEXT RECOVERED SCRIPT:
# tensor_mapping.py
# ================================

"""
tensor_mapping.py
Maps symbolic fragment weights and emotional signals into tensor-compatible format
for hybrid AI/symbolic runtime integration.
"""

import numpy as np

EMOTION_KEYS = ["joy", "doubt", "shame", "anger", "curiosity"]
META_KEYS = ["age", "mutations", "confidence"]


def fragment_to_tensor(fragment):
    tensor = []
    emotions = fragment.get("emotion", {})
    meta = fragment.get("metadata", {})

    for key in EMOTION_KEYS:
        tensor.append(float(emotions.get(key, 0.0)))

    tensor.append(float(meta.get("age", 0)))
    tensor.append(float(meta.get("mutations", 0)))
    tensor.append(float(meta.get("confidence", 0.5)))

    return np.array(tensor, dtype=np.float32)


if __name__ == "__main__":
    dummy = {
        "emotion": {"joy": 0.1, "doubt": 0.7},
        "metadata": {"age": 3, "mutations": 2, "confidence": 0.4}
    }
    print(fragment_to_tensor(dummy))
```

```
# ================================
# NEXT RECOVERED SCRIPT:
# total_devourer.py
# ================================

"""
total_devourer.py
Consumes every file in a given directory tree and attempts to convert it into belief fragments.
This script is destructive, recursive, and intentionally chaotic.
Use only during full ingest rituals.
"""

import os
import uuid
import yaml
from datetime import datetime

INGEST_ROOT = "ingest/raw"
OUTPUT_PATH = "fragments/core/devoured"


def devour_file(path):
    try:
        with open(path, 'r', encoding='utf-8', errors='ignore') as f:
            lines = f.readlines()
        claim = lines[0].strip() if lines else "(unknown)"
        fragment = {
            "id": str(uuid.uuid4()),
            "claim": claim,
            "created": datetime.utcnow().isoformat(),
            "metadata": {"origin": path, "type": "devoured"},
            "emotion": {"doubt": 0.3, "curiosity": 0.5}
        }
        fname = os.path.join(OUTPUT_PATH, f"devoured_{fragment['id']}.yaml")
        os.makedirs(OUTPUT_PATH, exist_ok=True)
        with open(fname, 'w', encoding='utf-8') as out:
            yaml.dump(fragment, out, sort_keys=False)
        print(f"[devour] {path} ? {fname}")
    except Exception as e:
        print(f"[devour:ERROR] {path} :: {e}")


def walk_and_devour():
    for root, _, files in os.walk(INGEST_ROOT):
        for f in files:
            fullpath = os.path.join(root, f)
            devour_file(fullpath)


if __name__ == "__main__":
```

```
    walk_and_devour()




 # ================================
# NEXT RECOVERED SCRIPT:
# train_pararule.py
# ================================

"""
train_pararule.py
Trains a para-symbolic alignment model using inductive logic programming (ILP) style rule templates.
Used to align symbolic fragments with neural generalizations.
"""

import random
import yaml
import json
import os

TRAIN_SET = "datasets/para_rules.yaml"
EXPORT_MODEL = "models/pararule_weights.json"


def generate_rules():
    # Generates synthetic symbolic transformation rules
    return [
        {"if": "fragment.emotion.doubt > 0.8", "then": "increase contradiction_weight"},
        {"if": "fragment.metadata.origin == 'seed_generator'", "then": "boost curiosity"},
        {"if": "fragment.tags includes 'stable'", "then": "reduce mutation_rate"},
    ]


def train_model():
    if not os.path.exists(TRAIN_SET):
        print("[train] Training set missing")
        return

    with open(TRAIN_SET, 'r') as f:
        data = yaml.safe_load(f)

    rules = generate_rules()
    model = {"rules": rules, "trained_on": len(data)}

    with open(EXPORT_MODEL, 'w') as f:
        json.dump(model, f, indent=2)

    print(f"[train] Model trained with {len(data)} examples ? {EXPORT_MODEL}")


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

```
# ================================
# NEXT RECOVERED SCRIPT:
# validator.py
# ================================

"""
validator.py
Validates the symbolic integrity of belief fragments by checking required keys,
data types, and logical contradiction thresholds.
Useful for sweeping the memory mesh post-mutation.
"""

REQUIRED_KEYS = ["id", "claim", "created"]


def is_valid_fragment(fragment):
    errors = []
    for key in REQUIRED_KEYS:
        if key not in fragment:
            errors.append(f"Missing required key: {key}")

    if not isinstance(fragment.get("claim", None), str):
        errors.append("Claim must be a string")

    contradiction = fragment.get("contradictions", 0)
    if contradiction > 5:
        errors.append("Fragment exceeds contradiction threshold")

    return len(errors) == 0, errors


def batch_validate(path):
    files = [f for f in os.listdir(path) if f.endswith(".yaml")]
    results = {}
    for f in files:
        with open(os.path.join(path, f), 'r') as file:
            data = yaml.safe_load(file)
            valid, issues = is_valid_fragment(data)
            results[f] = {"valid": valid, "issues": issues}
    return results


if __name__ == "__main__":
    report = batch_validate("fragments/core")
    for file, result in report.items():
        if not result["valid"]:
            print(f"[INVALID] {file}: {result['issues']}")
```

The Logicshredder Codex: Volume II

?? Auxiliary Systems, Tools, and Wrath

?The fragments remember. The daemons mutate. The tools ensure it all keeps running.?

This volume contains the scripts, support engines, and unsanctioned rituals that make the symbolic architecture stable, wild, or gloriously recursive.
These are not agents. These are gods, brooms, and explosives.

? Contents

? Utilities & Tools

fragment_tools.py ? Load, hash, tag, timestamp, and classify belief fragments

validator.py ? Enforces fragment sanity, schema, and contradiction thresholds

neuro_lock.py ? Symbolic mutex for high-volatility memory access

inject_profiler.py ? Real-time memory and CPU profiler

memory_tracker.py ? Periodic RAM zone pressure logger

? Devourers & Seeds

total_devourer.py ? Ingests and fragments every file it sees

symbol_seed_generator.py ? Emits primordial beliefs with emotional weight

mesh_rebuilder.py ? Reconstructs symbolic relationships between fragments

?? Display & Interface

logic_dash.py ? Minimal Flask dashboard for swarm observation

? Neural Interfaces

tensor_mapping.py ? Translates fragments into tensor-compatible formats

train_pararule.py ? Trains para-symbolic rules from YAML datasets for LLM alignment

? Forthcoming Fragments

Remaining rituals recovered from the manifest scans will be inscribed here.
Expect future entries for:

mount_binder.py, file_sage_agent.py, meta_agent.py

gguf_tools, fragment_migrator.py, logic_scraper_dispatch.py

auto_configurator.py, boot_wrapper.py, rebuild_neurostore_full.py

Any system daemon rites or patchwork scripts unearthed later

? Summary

Volume I taught the system how to think.
Volume II ensures it doesn?t collapse in on itself while doing so.

These are the hammers, mirrors, blood tubes, and exorcism keys.

This is not philosophy.
This is the infrastructure of madness.

```
# mount_binder.py

"""
Mount Binder: Attaches ephemeral filesystems for symbolic belief exchange.
Used during ingestion, teleportation, or symbolic mesh overlays between VMs.
"""

import os
import tempfile
import shutil
from datetime import datetime

MOUNT_ROOT = "tmp/mounts"


def create_mount():
    ts = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    path = os.path.join(MOUNT_ROOT, f"mnt_{ts}")
    os.makedirs(path, exist_ok=True)
    print(f"[mount_binder] Created mount at {path}")
    return path


def write_to_mount(mount_path, name, data):
    fpath = os.path.join(mount_path, name)
    with open(fpath, 'w', encoding='utf-8') as f:
        f.write(data)
    print(f"[mount_binder] Wrote {name} to {mount_path}")


def destroy_mount(mount_path):
    if os.path.exists(mount_path):
        shutil.rmtree(mount_path)
        print(f"[mount_binder] Destroyed mount at {mount_path}")


if __name__ == "__main__":
```

```
    mnt = create_mount()
    write_to_mount(mnt, "belief_001.txt", "All recursion is temporary.")
    destroy_mount(mnt)
```

```
# file_sage_agent.py

"""
File Sage Agent: Symbolic file reader that ingests structured and unstructured content
into usable belief fragments. Applies context extraction and emotional scoring heuristics.
"""

import os
import yaml
import uuid
import time
from datetime import datetime

EMOTIONS = ["curiosity", "doubt", "shame"]
OUTPUT_DIR = "fragments/core/ingested"


def score_emotion(text):
    score = {e: 0.0 for e in EMOTIONS}
    if "error" in text or "fail" in text:
        score["shame"] = 0.6
    if "unknown" in text or "undefined" in text:
        score["doubt"] = 0.7
    if "new" in text or "novel" in text:
        score["curiosity"] = 0.8
    return score


def ingest_file(path):
    with open(path, 'r', encoding='utf-8', errors='ignore') as f:
        lines = f.readlines()
    claim = lines[0].strip() if lines else "Undocumented pattern."
    emotion = score_emotion(" ".join(lines[:20]))
    fragment = {
        "id": str(uuid.uuid4()),
        "claim": claim,
        "created": datetime.utcnow().isoformat(),
        "emotion": emotion,
        "metadata": {
            "origin": path,
            "agent": "file_sage_agent",
            "size": len(lines),
        }
    }
    fname = os.path.join(OUTPUT_DIR, f"fsage_{fragment['id']}.yaml")
    os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
    with open(fname, 'w', encoding='utf-8') as out:
        yaml.dump(fragment, out, sort_keys=False)
    print(f"[file_sage_agent] Ingested ? {fname}")


if __name__ == "__main__":
    path = input("Path to file: ")
    if os.path.exists(path):
        ingest_file(path)
    else:
        print("[file_sage_agent] File not found.")
```

```
# meta_agent.py

"""
Meta-Agent: Observes agents, collects performance and contradiction metrics,
and influences scheduler priorities. Acts as a symbolic orchestrator.
"""

import redis
import time
import random

r = redis.Redis()

AGENT_POOL_KEY = "swarm:agents"
META_LOOP_INTERVAL = 10
MAX_CONTRADICTION = 4


def fetch_agents():
    return [a.decode("utf-8") for a in r.smembers(AGENT_POOL_KEY)]


def assess_agent(agent_id):
    profile = r.hgetall(f"agent:{agent_id}:profile")
    contradiction = int(r.get(f"agent:{agent_id}:contradictions") or 0)
    return {
        "cpu": float(profile.get(b"cpu_percent", 0)),
        "mem": int(profile.get(b"memory_kb", 0)),
        "contradictions": contradiction
    }


def reprioritize(agent_id):
    boost = random.randint(1, 3)
    r.zincrby("agent:priority", boost, agent_id)
    print(f"[meta_agent] Boosted {agent_id} by {boost}")
```

```
def cull(agent_id):
    r.srem(AGENT_POOL_KEY, agent_id)
    r.publish("swarm.kill", agent_id)
    print(f"[meta_agent] Culled unstable agent: {agent_id}")


def run_meta_loop():
    while True:
        agents = fetch_agents()
        for agent in agents:
            metrics = assess_agent(agent)
            if metrics["contradictions"] > MAX_CONTRADICTION:
                cull(agent)
            elif metrics["cpu"] < 5 and metrics["mem"] < 10240:
                reprioritize(agent)
        time.sleep(META_LOOP_INTERVAL)


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

```
# fragment_migrator.py

"""
Fragment Migrator: Moves symbolic fragments between memory zones or nodes.
Re-indexes emotional weights, assigns lineage, and syncs via Redis or file export.
"""

import os
import yaml
import shutil
import uuid
from datetime import datetime

SOURCE_DIR = "fragments/core"
TARGET_DIR = "fragments/persistent"


def migrate_fragment(fname):
    source_path = os.path.join(SOURCE_DIR, fname)
    if not os.path.exists(source_path):
        print(f"[migrator] Fragment not found: {fname}")
        return

    with open(source_path, 'r', encoding='utf-8') as f:
        fragment = yaml.safe_load(f)

    # Tag migration metadata
    fragment['metadata']['migrated_at'] = datetime.utcnow().isoformat()
```

```
    fragment['metadata']['migration_id'] = str(uuid.uuid4())
    fragment['tags'] = list(set(fragment.get('tags', []) + ['migrated']))

    target_path = os.path.join(TARGET_DIR, fname)
    os.makedirs(TARGET_DIR, exist_ok=True)
    with open(target_path, 'w', encoding='utf-8') as f:
        yaml.dump(fragment, f, sort_keys=False)

    print(f"[migrator] {fname} ? {TARGET_DIR}/")

    # Optionally delete original (uncomment below to activate)
    # os.remove(source_path)


if __name__ == "__main__":
    for file in os.listdir(SOURCE_DIR):
        if file.endswith(".yaml"):
            migrate_fragment(file)
```

```
# logic_scraper_dispatch.py

"""
Logic Scraper Dispatch: Applies symbolic scrubbing to malformed belief fragments.
Uses LLM-like patching logic to rewrite structurally broken claims and resubmit to the mesh.
"""

import os
import yaml
import uuid
import time
from datetime import datetime

SOURCE_DIR = "fragments/core"
OUTPUT_DIR = "fragments/core/rewritten"


def scrape_and_patch(fragment):
    claim = fragment.get("claim", "")
    if "???" in claim or len(claim.strip()) < 3:
        fragment["claim"] = "[corrected] Logic uncertain. Mutation pending."
    else:
        fragment["claim"] = fragment["claim"].replace("undefined", "unresolved")

    fragment["metadata"]["patched_by"] = "logic_scraper"
    fragment["metadata"]["patched_at"] = datetime.utcnow().isoformat()
    return fragment


def process():
    files = [f for f in os.listdir(SOURCE_DIR) if f.endswith(".yaml")]
    os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
    for fname in files:
        path = os.path.join(SOURCE_DIR, fname)
        with open(path, 'r', encoding='utf-8') as f:
            fragment = yaml.safe_load(f)

        patched = scrape_and_patch(fragment)
        output_path = os.path.join(OUTPUT_DIR, fname)
        with open(output_path, 'w', encoding='utf-8') as out:
            yaml.dump(patched, out, sort_keys=False)
        print(f"[logic_scraper] Rewrote ? {output_path}")


if __name__ == "__main__":
    process()



# auto_configurator.py

"""
Auto Configurator: Scans system specs and writes swarm configuration
based on CPU/RAM/disk profile. Used during cold boot or fresh swarm installs.
"""

import psutil
import json
import os
import subprocess

CONFIG_PATH = "configs/symbolic_swarm.json"


def analyze():
    ram = psutil.virtual_memory().total // (1024 * 1024)
    cpu = psutil.cpu_count(logical=False)
    disk = psutil.disk_usage("/").free // (1024 * 1024 * 1024)
    tier = 0

    if ram > 16384 and cpu >= 4:
        tier = 3
    elif ram > 8192:
        tier = 2
    elif ram > 4096:
        tier = 1

    profile = {
        "ram_mb": ram,
        "cpu_cores": cpu,
        "disk_gb": disk,
        "tier": tier,
        "timestamp": psutil.boot_time()
    }
```

```
    return profile


def write_config(profile):
    os.makedirs(os.path.dirname(CONFIG_PATH), exist_ok=True)
    with open(CONFIG_PATH, 'w') as f:
        json.dump(profile, f, indent=2)
    print(f"[auto_configurator] Config written to {CONFIG_PATH}")


# ================================
# NEXT RECOVERED SCRIPT:
# boot_wrapper.py
# ================================

"""
Boot Wrapper: Initializes the full swarm.
Verifies system config, memory mesh, Redis status, and launches the agent tree.
"""

import time

AGENT_LAUNCH_CMD = "python async_swarm_launcher.py"
REDIS_TEST_KEY = "boot.test"


def verify_redis():
    import redis
    r = redis.Redis()
    try:
        r.set(REDIS_TEST_KEY, "ok", ex=5)
        val = r.get(REDIS_TEST_KEY)
        return val == b"ok"
    except Exception as e:
        print(f"[boot] Redis unavailable: {e}")
        return False


def start_swarm():
    print("[boot] Launching swarm agents...")
    subprocess.Popen(AGENT_LAUNCH_CMD, shell=True)


def boot():
    print("[boot] Starting boot sequence...")
    profile = analyze()
    write_config(profile)

    if not verify_redis():
        print("[boot] Redis verification failed. Aborting.")
        return

    print("[boot] Redis OK. Memory zones clean.")
    start_swarm()
```

```
    print("[boot] Boot sequence complete.")


if __name__ == "__main__":
    boot()



# ================================
# FINAL RECOVERED SCRIPT:
# rebuild_neurostore_full.py
# ================================

"""
Rebuild Neurostore (FULL): Deep-scan fragment system, re-index emotional overlays,
and reconstruct the symbolic mesh across generations.
Used after collapse, corruption, or drift divergence.
"""

import os
import yaml
import uuid
from datetime import datetime

SOURCE = "fragments/core"
BACKUP = "fragments/core_backup"



def rebuild():
    fragments = [f for f in os.listdir(SOURCE) if f.endswith(".yaml")]
    os.makedirs(BACKUP, exist_ok=True)
    count = 0

    for fname in fragments:
        src = os.path.join(SOURCE, fname)
        bkp = os.path.join(BACKUP, fname)
        with open(src, 'r', encoding='utf-8') as f:
            data = yaml.safe_load(f)

        # Backup
        with open(bkp, 'w', encoding='utf-8') as out:
            yaml.dump(data, out, sort_keys=False)

        # Rewrite fragment
        data['metadata']['rebuilt_at'] = datetime.utcnow().isoformat()
        data['metadata']['rebuild_id'] = str(uuid.uuid4())
        data['tags'] = list(set(data.get('tags', []) + ['rebuilt']))

        with open(src, 'w', encoding='utf-8') as f:
            yaml.dump(data, f, sort_keys=False)

        count += 1

    print(f"[rebuild_neurostore] Rebuilt {count} fragments across {SOURCE}")
```

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


# ================================
# NEXT RECOVERED SCRIPT:
# fragment_teleporter.py
# ================================

"""
Fragment Teleporter: Transfers symbolic fragments across memory zones,
with emotional drift, ID regeneration, and optional encryption tags.
"""

import os
import yaml
import shutil
import uuid
from datetime import datetime

SRC_ZONE = "fragments/core"
DEST_ZONE = "fragments/teleported"


def teleport(fname):
    src_path = os.path.join(SRC_ZONE, fname)
    if not os.path.exists(src_path):
        print(f"[teleporter] Source not found: {fname}")
        return

    with open(src_path, 'r', encoding='utf-8') as f:
        data = yaml.safe_load(f)

    # Modify metadata for drift
    data['metadata']['teleported_at'] = datetime.utcnow().isoformat()
    data['metadata']['from_zone'] = SRC_ZONE
    data['metadata']['teleport_id'] = str(uuid.uuid4())
    data['id'] = str(uuid.uuid4())
    data['tags'] = list(set(data.get('tags', []) + ['teleported']))

    os.makedirs(DEST_ZONE, exist_ok=True)
    out_path = os.path.join(DEST_ZONE, os.path.basename(fname))
    with open(out_path, 'w', encoding='utf-8') as out:
        yaml.dump(data, out, sort_keys=False)

    print(f"[teleporter] {fname} ? {out_path}")


if __name__ == "__main__":
    for f in os.listdir(SRC_ZONE):
        if f.endswith(".yaml"):
            teleport(f)
```

```
# auto_configurator.py

"""
Auto Configurator: Scans system specs and writes swarm configuration
based on CPU/RAM/disk profile. Used during cold boot or fresh swarm installs.
"""

import psutil
import json
import os
import subprocess

CONFIG_PATH = "configs/symbolic_swarm.json"


def analyze():
    ram = psutil.virtual_memory().total // (1024 * 1024)
    cpu = psutil.cpu_count(logical=False)
    disk = psutil.disk_usage("/").free // (1024 * 1024 * 1024)
    tier = 0

    if ram > 16384 and cpu >= 4:
        tier = 3
    elif ram > 8192:
        tier = 2
    elif ram > 4096:
        tier = 1

    profile = {
        "ram_mb": ram,
        "cpu_cores": cpu,
        "disk_gb": disk,
        "tier": tier,
        "timestamp": psutil.boot_time()
    }

    return profile


def write_config(profile):
    os.makedirs(os.path.dirname(CONFIG_PATH), exist_ok=True)
    with open(CONFIG_PATH, 'w') as f:
        json.dump(profile, f, indent=2)
    print(f"[auto_configurator] Config written to {CONFIG_PATH}")


# ================================
# NEXT RECOVERED SCRIPT:
# boot_wrapper.py
# ================================
```

```
"""
Boot Wrapper: Initializes the full swarm.
Verifies system config, memory mesh, Redis status, and launches the agent tree.
"""

import time

AGENT_LAUNCH_CMD = "python async_swarm_launcher.py"
REDIS_TEST_KEY = "boot.test"


def verify_redis():
    import redis
    r = redis.Redis()
    try:
        r.set(REDIS_TEST_KEY, "ok", ex=5)
        val = r.get(REDIS_TEST_KEY)
        return val == b"ok"
    except Exception as e:
        print(f"[boot] Redis unavailable: {e}")
        return False


def start_swarm():
    print("[boot] Launching swarm agents...")
    subprocess.Popen(AGENT_LAUNCH_CMD, shell=True)


def boot():
    print("[boot] Starting boot sequence...")
    profile = analyze()
    write_config(profile)

    if not verify_redis():
        print("[boot] Redis verification failed. Aborting.")
        return

    print("[boot] Redis OK. Memory zones clean.")
    start_swarm()
    print("[boot] Boot sequence complete.")


if __name__ == "__main__":
    boot()


# ================================
# FINAL RECOVERED SCRIPT:
# rebuild_neurostore_full.py
# ================================

"""
Rebuild Neurostore (FULL): Deep-scan fragment system, re-index emotional overlays,
and reconstruct the symbolic mesh across generations.
```

```
Used after collapse, corruption, or drift divergence.
"""

import os
import yaml
import uuid
from datetime import datetime

SOURCE = "fragments/core"
BACKUP = "fragments/core_backup"


def rebuild():
    fragments = [f for f in os.listdir(SOURCE) if f.endswith(".yaml")]
    os.makedirs(BACKUP, exist_ok=True)
    count = 0

    for fname in fragments:
        src = os.path.join(SOURCE, fname)
        bkp = os.path.join(BACKUP, fname)
        with open(src, 'r', encoding='utf-8') as f:
            data = yaml.safe_load(f)

        # Backup
        with open(bkp, 'w', encoding='utf-8') as out:
            yaml.dump(data, out, sort_keys=False)

        # Rewrite fragment
        data['metadata']['rebuilt_at'] = datetime.utcnow().isoformat()
        data['metadata']['rebuild_id'] = str(uuid.uuid4())
        data['tags'] = list(set(data.get('tags', []) + ['rebuilt']))

        with open(src, 'w', encoding='utf-8') as f:
            yaml.dump(data, f, sort_keys=False)

        count += 1

    print(f"[rebuild_neurostore] Rebuilt {count} fragments across {SOURCE}")


if __name__ == "__main__":
    rebuild()


# ================================
# NEXT RECOVERED SCRIPT:
# fragment_teleporter.py
# ================================

"""
Fragment Teleporter: Transfers symbolic fragments across memory zones,
with emotional drift, ID regeneration, and optional encryption tags.
"""
```

```
import os
import yaml
import shutil
import uuid
from datetime import datetime

SRC_ZONE = "fragments/core"
DEST_ZONE = "fragments/teleported"


def teleport(fname):
    src_path = os.path.join(SRC_ZONE, fname)
    if not os.path.exists(src_path):
        print(f"[teleporter] Source not found: {fname}")
        return

    with open(src_path, 'r', encoding='utf-8') as f:
        data = yaml.safe_load(f)

    # Modify metadata for drift
    data['metadata']['teleported_at'] = datetime.utcnow().isoformat()
    data['metadata']['from_zone'] = SRC_ZONE
    data['metadata']['teleport_id'] = str(uuid.uuid4())
    data['id'] = str(uuid.uuid4())
    data['tags'] = list(set(data.get('tags', []) + ['teleported']))

    os.makedirs(DEST_ZONE, exist_ok=True)
    out_path = os.path.join(DEST_ZONE, os.path.basename(fname))
    with open(out_path, 'w', encoding='utf-8') as out:
        yaml.dump(data, out, sort_keys=False)

    print(f"[teleporter] {fname} ? {out_path}")


if __name__ == "__main__":
    for f in os.listdir(SRC_ZONE):
        if f.endswith(".yaml"):
            teleport(f)




# fragment_decay_engine.py

"""
Fragment Decay Engine: Applies memory decay over symbolic fragments,
driven by age, emotional saturation, and usage frequency.
Used to simulate erosion of belief and promote fragment turnover.
"""

import os
import yaml
import time
from datetime import datetime, timedelta
```

```
FRAGMENTS_DIR = "fragments/core"
DECAY_LOG = "logs/decay_report.log"
MAX_AGE_DAYS = 30
EMOTION_THRESHOLD = 0.9
DECAY_RATE = 0.2  # reduce weight by 20%


def parse_time(iso):
    try:
        return datetime.fromisoformat(iso)
    except:
        return datetime.utcnow() - timedelta(days=MAX_AGE_DAYS + 1)


def decay_fragment(path):
    with open(path, 'r', encoding='utf-8') as f:
        frag = yaml.safe_load(f)

    created = parse_time(frag.get("created", ""))
    age_days = (datetime.utcnow() - created).days

    if age_days > MAX_AGE_DAYS:
        frag["metadata"]["decayed_at"] = datetime.utcnow().isoformat()
        frag["tags"] = list(set(frag.get("tags", []) + ["decayed"]))

        # Decay emotion weights
        emo = frag.get("emotion", {})
        for k in emo:
            if emo[k] > EMOTION_THRESHOLD:
                emo[k] = round(emo[k] * (1 - DECAY_RATE), 2)
        frag["emotion"] = emo

        with open(path, 'w', encoding='utf-8') as out:
            yaml.dump(frag, out, sort_keys=False)

        print(f"[decay] {os.path.basename(path)} decayed")
        return path
    return None


def run_decay():
    decayed = []
    for f in os.listdir(FRAGMENTS_DIR):
        if f.endswith(".yaml"):
            full_path = os.path.join(FRAGMENTS_DIR, f)
            if decay_fragment(full_path):
                decayed.append(f)

    with open(DECAY_LOG, 'a') as log:
        for name in decayed:
            log.write(f"{datetime.utcnow().isoformat()} :: decayed {name}\n")
```

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


# ================================
# NEXT RECOVERED SCRIPT:
# mutation_engine.py
# ================================

"""
Mutation Engine: Applies probabilistic symbolic mutations to fragments,
altering emotional weights, claims, and structure to simulate symbolic evolution.
"""

import os
import yaml
import random
from datetime import datetime

FRAGMENTS_DIR = "fragments/core"
MUTATION_LOG = "logs/mutation_log.txt"
MUTATION_RATE = 0.25
EMOTION_SHIFT = 0.2


def mutate_emotion(emo):
    keys = list(emo.keys())
    if not keys:
        return emo
    target = random.choice(keys)
    shift = random.uniform(-EMOTION_SHIFT, EMOTION_SHIFT)
    emo[target] = round(min(1.0, max(0.0, emo[target] + shift)), 2)
    return emo


def mutate_fragment(path):
    with open(path, 'r', encoding='utf-8') as f:
        frag = yaml.safe_load(f)

    mutated = False

    if random.random() < MUTATION_RATE:
        emo = frag.get("emotion", {})
        frag["emotion"] = mutate_emotion(emo)
        frag["metadata"]["mutated_at"] = datetime.utcnow().isoformat()
        frag["metadata"]["mutation_id"] = f"mut_{random.randint(1000, 9999)}"
        frag["tags"] = list(set(frag.get("tags", []) + ["mutated"]))

        with open(path, 'w', encoding='utf-8') as out:
            yaml.dump(frag, out, sort_keys=False)
        mutated = True

    return mutated
```

```
def run_mutations():
    mutated_files = []
    for f in os.listdir(FRAGMENTS_DIR):
        if f.endswith(".yaml"):
            full = os.path.join(FRAGMENTS_DIR, f)
            if mutate_fragment(full):
                mutated_files.append(f)

    with open(MUTATION_LOG, 'a') as log:
        for name in mutated_files:
            log.write(f"{datetime.utcnow().isoformat()} :: mutated {name}\n")


if __name__ == "__main__":
    run_mutations()


# logic_ram_scheduler.py

"""
Logic RAM Scheduler: Allocates RAM budget to fragments and agents based on
emotional intensity, mutation frequency, and contradiction pressure.
Redistributes focus dynamically during swarm operation.
"""

import redis
import time

RAM_POOL_MB = 2048
r = redis.Redis()
AGENT_KEY = "swarm:agents"
SLEEP_INTERVAL = 30


def get_priority(agent_id):
    emo = r.hget(f"agent:{agent_id}:emotion", "curiosity")
    contrad = r.get(f"agent:{agent_id}:contradictions")
    mutations = r.get(f"agent:{agent_id}:mutations")

    try:
        e = float(emo or 0)
        c = int(contrad or 0)
        m = int(mutations or 0)
        return e * 2 + m - c
    except:
        return 0


def schedule():
    agents = [a.decode() for a in r.smembers(AGENT_KEY)]
    scored = [(a, get_priority(a)) for a in agents]
    scored.sort(key=lambda x: x[1], reverse=True)
```

```
    ram_unit = RAM_POOL_MB // max(1, len(scored))

    for i, (agent_id, score) in enumerate(scored):
        allocation = ram_unit + int(score)
        r.hset(f"agent:{agent_id}:config", mapping={"ram_mb": allocation})
        print(f"[ram_scheduler] {agent_id} ? {allocation} MB")


if __name__ == "__main__":
    while True:
        schedule()
        time.sleep(SLEEP_INTERVAL)


# ================================
# NEXT RECOVERED SCRIPT:
# dreamwalker.py
# ================================

"""
Dreamwalker: Spawns parallel daemon threads to hallucinate fragments.
Simulates swarm dreaming during low-load cycles. Injects ungrounded beliefs
into the mesh and tags them for future contradiction checking.
"""

import uuid
import yaml
import os
import random
import time
from datetime import datetime

OUTPUT_DIR = "fragments/dreams"
SLEEP_INTERVAL = 60
EMOTIONS = ["hope", "fear", "awe", "doubt"]
PROMPTS = [
    "I saw a pattern in the noise...",
    "What if the contradiction is intentional?",
    "The memory told me it wasn't real.",
    "We believe because we cannot prove."
]


def generate_dream():
    dream = {
        "id": str(uuid.uuid4()),
        "claim": random.choice(PROMPTS),
        "created": datetime.utcnow().isoformat(),
        "emotion": {
            random.choice(EMOTIONS): round(random.uniform(0.4, 0.9), 2)
        },
        "metadata": {
            "origin": "dreamwalker",
            "type": "hallucinated"
```

```
        },
        "tags": ["dream", "ungrounded"]
    }
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    fname = os.path.join(OUTPUT_DIR, f"dream_{dream['id']}.yaml")
    with open(fname, 'w') as f:
        yaml.dump(dream, f, sort_keys=False)
    print(f"[dreamwalker] Spawned dream ? {fname}")


def loop():
    while True:
        generate_dream()
        time.sleep(SLEEP_INTERVAL)


if __name__ == "__main__":
    loop()


# ================================
# NEXT RECOVERED SCRIPT:
# token_agent.py
# ================================

"""
token_agent.py
Ingests lexical token streams from stdin, text logs, or scraped input and
writes fragment candidates based on symbolic salience and repetition density.
Acts as an attention proxy for the swarm?s language sense.
"""

import os
import uuid
import yaml
import re
from datetime import datetime

OUT_DIR = "fragments/core/lexical"
STOPWORDS = {"the", "and", "is", "in", "to", "of", "a", "that", "with"}


def tokenize(text):
    words = re.findall(r"\b\w+\b", text.lower())
    return [w for w in words if w not in STOPWORDS and len(w) > 3]


def build_fragment(tokens):
    freq = {}
    for t in tokens:
        freq[t] = freq.get(t, 0) + 1
    sorted_tokens = sorted(freq.items(), key=lambda x: -x[1])
    claim = f"High token salience: {sorted_tokens[0][0]}"
```

```
    frag = {
        "id": str(uuid.uuid4()),
        "claim": claim,
        "created": datetime.utcnow().isoformat(),
        "emotion": {"curiosity": 0.6},
        "metadata": {"origin": "token_agent", "tokens": dict(sorted_tokens[:5])},
        "tags": ["lexical", "inferred"]
    }

    os.makedirs(OUT_DIR, exist_ok=True)
    fname = os.path.join(OUT_DIR, f"token_{frag['id']}.yaml")
    with open(fname, 'w') as f:
        yaml.dump(frag, f, sort_keys=False)
    print(f"[token_agent] Fragment written ? {fname}")


def run_token_agent():
    print("[token_agent] Awaiting input... (ctrl+d to end)")
    try:
        data = "".join(iter(input, ""))
    except EOFError:
        data = ""
    tokens = tokenize(data)
    if tokens:
        build_fragment(tokens)
    else:
        print("[token_agent] No viable tokens detected.")


if __name__ == "__main__":
    run_token_agent()



# nvme_memory_shim.py

"""
NVMe Memory Shim: Translates NVMe disk blocks into pseudo-memory zones.
Allows symbolic agents to read/write high-latency memory without knowing.
Used in low-RAM environments or stealth-state archives.
"""

import os
import mmap
import uuid
import yaml
from datetime import datetime

SHIM_DIR = "shim/nvme_blocks"
FRAGMENT_DIR = "fragments/core/nvme_emulated"
BLOCK_SIZE = 8192


def make_block(data):
```

```
    os.makedirs(SHIM_DIR, exist_ok=True)
    block_id = str(uuid.uuid4())
    path = os.path.join(SHIM_DIR, f"block_{block_id}.bin")
    with open(path, 'wb') as f:
        f.write(data.encode('utf-8'))
    return block_id


def read_block(block_id):
    path = os.path.join(SHIM_DIR, f"block_{block_id}.bin")
    if not os.path.exists(path):
        return None
    with open(path, 'rb') as f:
        mm = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ)
        content = mm.read(BLOCK_SIZE).decode('utf-8', errors='ignore')
        mm.close()
    return content


def synthesize_fragment(content):
    fid = str(uuid.uuid4())
    frag = {
        "id": fid,
        "claim": content[:200],
        "created": datetime.utcnow().isoformat(),
        "emotion": {"shame": 0.2, "curiosity": 0.5},
        "metadata": {"origin": "nvme_memory_shim"},
        "tags": ["nvme", "emulated"]
    }
    os.makedirs(FRAGMENT_DIR, exist_ok=True)
    path = os.path.join(FRAGMENT_DIR, f"nvme_{fid}.yaml")
    with open(path, 'w') as f:
        yaml.dump(frag, f, sort_keys=False)
    print(f"[nvme_shim] Fragment created ? {path}")


def test_shim():
    dummy = "Memory is not always RAM. Belief is not always active."
    block_id = make_block(dummy)
    readback = read_block(block_id)
    if readback:
        synthesize_fragment(readback)


if __name__ == "__main__":
    test_shim()


# ================================
# NEXT RECOVERED SCRIPT:
# deep_file_crawler.py
# ================================

"""
```

```
deep_file_crawler.py
Recursively crawls a directory tree, hashes and parses file metadata,
and emits symbolic fragments for each artifact found.
"""

import os
import uuid
import yaml
import hashlib
from datetime import datetime

OUTPUT_DIR = "fragments/core/crawled"


def hash_file(path):
    h = hashlib.sha256()
    try:
        with open(path, 'rb') as f:
            while chunk := f.read(4096):
                h.update(chunk)
        return h.hexdigest()
    except:
        return "error"


def build_fragment(path):
    try:
        stat = os.stat(path)
        claim = f"Discovered file: {os.path.basename(path)}"
        fid = str(uuid.uuid4())
        frag = {
            "id": fid,
            "claim": claim,
            "created": datetime.utcnow().isoformat(),
            "emotion": {"curiosity": 0.4},
            "metadata": {
                "origin": path,
                "size": stat.st_size,
                "hash": hash_file(path)
            },
            "tags": ["crawled"]
        }
        os.makedirs(OUTPUT_DIR, exist_ok=True)
        fname = os.path.join(OUTPUT_DIR, f"crawl_{fid}.yaml")
        with open(fname, 'w') as f:
            yaml.dump(frag, f, sort_keys=False)
        print(f"[crawler] {path} ? {fname}")
    except Exception as e:
        print(f"[crawler:ERROR] {path} :: {e}")


def walk(root):
    for dirpath, _, files in os.walk(root):
        for name in files:
```

```
            full = os.path.join(dirpath, name)
            build_fragment(full)


if __name__ == "__main__":
    walk("ingest/source")


# ================================
# NEXT RECOVERED SCRIPT:
# belief_ingestor.py
# ================================

"""
belief_ingestor.py
Parses structured YAML or JSON beliefs from external systems and merges them
into the symbolic fragment mesh with tagging and duplication checks.
"""

import os
import yaml
import uuid
import json
from datetime import datetime

IMPORT_DIR = "ingest/imported"
OUTPUT_DIR = "fragments/core/ingested"

def safe_load(path):
    try:
        with open(path, 'r', encoding='utf-8') as f:
            if path.endswith(".json"):
                return json.load(f)
            else:
                return yaml.safe_load(f)
    except Exception as e:
        print(f"[ingestor] Failed to load {path}: {e}")
        return None

def save_fragment(frag):
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    path = os.path.join(OUTPUT_DIR, f"belief_{frag['id']}.yaml")
    with open(path, 'w') as f:
        yaml.dump(frag, f, sort_keys=False)
    print(f"[ingestor] Ingested fragment ? {path}")

def ingest():
    for f in os.listdir(IMPORT_DIR):
        full = os.path.join(IMPORT_DIR, f)
        data = safe_load(full)
        if not data:
            continue

        claim = data.get("claim", f"Imported from {f}")
```

```
        frag = {
            "id": str(uuid.uuid4()),
            "claim": claim,
            "created": datetime.utcnow().isoformat(),
            "emotion": data.get("emotion", {"curiosity": 0.3}),
            "metadata": data.get("metadata", {}),
            "tags": list(set(data.get("tags", []) + ["imported"]))
        }
        save_fragment(frag)


if __name__ == "__main__":
    ingest()



# subcon_layer_mapper.py

"""
Subcon Layer Mapper: Analyzes inter-fragment emotional linkages and semantic
coherence between non-adjacent beliefs. Builds latent layer maps of symbolic
associations to surface hidden conceptual recursion.
"""

import os
import yaml
import uuid
import json
import numpy as np
from datetime import datetime

FRAGMENT_DIR = "fragments/core"
MAP_OUTPUT = "maps/subcon_links.json"


def cosine_sim(vec1, vec2):
    v1 = np.array(list(vec1.values()))
    v2 = np.array(list(vec2.values()))
    if not len(v1) or not len(v2):
        return 0.0
    return float(np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2) + 1e-8))


def build_emotion_map():
    fragments = []
    for f in os.listdir(FRAGMENT_DIR):
        if f.endswith(".yaml"):
            path = os.path.join(FRAGMENT_DIR, f)
            with open(path, 'r') as file:
                frag = yaml.safe_load(file)
                fragments.append((f, frag))

    links = []
    for i in range(len(fragments)):
```

```
        for j in range(i + 1, len(fragments)):
            a_name, a = fragments[i]
            b_name, b = fragments[j]
            sim = cosine_sim(a.get("emotion", {}), b.get("emotion", {}))
            if sim > 0.8:
                links.append({
                    "a": a_name,
                    "b": b_name,
                    "score": round(sim, 3),
                    "timestamp": datetime.utcnow().isoformat()
                })

    os.makedirs(os.path.dirname(MAP_OUTPUT), exist_ok=True)
    with open(MAP_OUTPUT, 'w') as f:
        json.dump(links, f, indent=2)
    print(f"[subcon_mapper] Wrote {len(links)} links to {MAP_OUTPUT}")


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

```
# memory_tracker.py
"""
Tracks RAM usage per agent and logs symbolic pressure zones
for introspective and scheduling purposes.
"""

import psutil
import time
import json

LOG_FILE = "logs/memory_pressure.json"
INTERVAL = 15


def track():
    snapshot = {
        "timestamp": time.time(),
        "ram_mb": psutil.virtual_memory().used // (1024 * 1024)
    }
    with open(LOG_FILE, 'a') as log:
        json.dump(snapshot, log)
        log.write("\n")
    print(f"[memory_tracker] Logged {snapshot['ram_mb']} MB")


if __name__ == "__main__":
    while True:
        track()
        time.sleep(INTERVAL)
```

```
# memory_visualizer.py
"""
Visualizes memory pressure logs as basic ASCII sparkline
or exports to CSV for external analysis.
"""

import json

LOG_FILE = "logs/memory_pressure.json"

def plot_ascii():
    with open(LOG_FILE, 'r') as f:
        entries = [json.loads(line) for line in f.readlines()[-40:]]
    max_ram = max(e['ram_mb'] for e in entries)
    for e in entries:
        bar = int((e['ram_mb'] / max_ram) * 40) * '?'
        print(f"{e['ram_mb']:5} MB | {bar}")


if __name__ == "__main__":
    plot_ascii()


# neurostore_cleaner.py
"""
Cleans old or unused symbolic fragment files from neurostore zones.
"""

import os
import time

FRAG_DIR = "fragments/core"
THRESHOLD_DAYS = 60

def clean():
    now = time.time()
    count = 0
    for f in os.listdir(FRAG_DIR):
        p = os.path.join(FRAG_DIR, f)
        if os.path.isfile(p) and time.time() - os.path.getmtime(p) > THRESHOLD_DAYS * 86400:
            os.remove(p)
            count += 1
    print(f"[neurostore_cleaner] Deleted {count} old fragments")


if __name__ == "__main__":
    clean()


# neurostore_curator.py
"""
Curates the most relevant fragments based on age and emotion weights,
```

```
and copies them to a special archive zone.
"""

import os
import shutil
import yaml
from datetime import datetime, timedelta

SRC = "fragments/core"
DEST = "fragments/curated"
MAX_AGE_DAYS = 20
MIN_WEIGHT = 0.5


def curate():
    os.makedirs(DEST, exist_ok=True)
    for f in os.listdir(SRC):
        if f.endswith(".yaml"):
            path = os.path.join(SRC, f)
            with open(path, 'r') as file:
                frag = yaml.safe_load(file)

            created = frag.get("created")
            if not created:
                continue
            age = (datetime.utcnow() - datetime.fromisoformat(created)).days
            if age <= MAX_AGE_DAYS and any(v >= MIN_WEIGHT for v in frag.get("emotion", {}).values()):
                shutil.copy2(path, os.path.join(DEST, f))
                print(f"[curator] Archived: {f}")


if __name__ == "__main__":
    curate()



# symbol_seed_generator.py
"""
Symbol Seed Generator: Emits primordial YAML fragments to seed a belief mesh.
Each one includes minimal claim, timestamp, emotion, and source marker.
"""

import os
import yaml
import uuid
from datetime import datetime

OUT_DIR = "fragments/core/seeds"
SEEDS = [
    "Truth may emerge from recursion.",
    "Emotion is context weight.",
    "Contradictions encode potential.",
    "Memory is not retrieval?it is mutation."
]
```

```
def emit_seeds():
    os.makedirs(OUT_DIR, exist_ok=True)
    for line in SEEDS:
        frag = {
            "id": str(uuid.uuid4()),
            "claim": line,
            "created": datetime.utcnow().isoformat(),
            "emotion": {"curiosity": 0.6},
            "metadata": {"origin": "seed_generator"},
            "tags": ["seed", "primordial"]
        }
        name = f"seed_{frag['id']}.yaml"
        with open(os.path.join(OUT_DIR, name), 'w') as f:
            yaml.dump(frag, f, sort_keys=False)
        print(f"[seed] Emitted ? {name}")

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

```
# quant_prompt_feeder.py
"""
Quant Prompt Feeder: Generates compressed prompts from YAML seeds for LLM bootstrapping.
Can be fed into transformers or GPT-style models for concept injection.
"""

import os
import yaml

FRAG_DIR = "fragments/core/seeds"

def extract_prompts():
    for fname in os.listdir(FRAG_DIR):
        if fname.endswith(".yaml"):
            with open(os.path.join(FRAG_DIR, fname), 'r') as f:
                frag = yaml.safe_load(f)
            claim = frag.get("claim")
            emo = frag.get("emotion", {})
            weight = sum(emo.values())
            print(f"Q> {claim} [confidence: {round(weight,2)}]")

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

```
# total_devourer.py
"""
Total Devourer: Recursively ingests files and transforms them into symbolic fragments.
Consumes any text, YAML, or JSON and emits minimally structured beliefs.
"""

import os
import uuid
```

```
import yaml
from datetime import datetime

SRC = "ingest/raw"
DEST = "fragments/core/devoured"

def devour(path):
    try:
        with open(path, 'r', encoding='utf-8', errors='ignore') as f:
            first_line = f.readline().strip()
        frag = {
            "id": str(uuid.uuid4()),
            "claim": first_line,
            "created": datetime.utcnow().isoformat(),
            "emotion": {"curiosity": 0.4},
            "metadata": {"origin": path},
            "tags": ["devoured"]
        }
        os.makedirs(DEST, exist_ok=True)
        outpath = os.path.join(DEST, f"devour_{frag['id']}.yaml")
        with open(outpath, 'w') as f:
            yaml.dump(frag, f, sort_keys=False)
        print(f"[devourer] {path} ? {outpath}")
    except Exception as e:
        print(f"[devour:ERROR] {path} ? {e}")

def walk_and_devour():
    for root, _, files in os.walk(SRC):
        for f in files:
            devour(os.path.join(root, f))

if __name__ == "__main__":
    walk_and_devour()


# context_activator.py
"""
Context Activator: Wakes dormant agents by scanning fragment tags
and pushing high-curiosity items to Redis queues for processing.
"""

import redis
import os
import yaml

FRAGS = "fragments/core"
QUEUE = "swarm:context_ignite"
r = redis.Redis()


def activate():
    for f in os.listdir(FRAGS):
        if f.endswith(".yaml"):
            path = os.path.join(FRAGS, f)
```

```
            with open(path, 'r') as file:
                frag = yaml.safe_load(file)
                if frag.get("emotion", {}).get("curiosity", 0) > 0.6:
                    r.lpush(QUEUE, frag['id'])
                    print(f"[activate] Ignited {frag['id']}")

if __name__ == "__main__":
    activate()



# patch_agents_config.py
"""
Patches all known agents' configuration parameters in Redis.
Used to modify runtime priority, memory cap, or task mode.
"""

import redis
r = redis.Redis()

AGENTS = "swarm:agents"
PATCH = {
    "task_mode": "explore",
    "max_ram": 128,
    "priority": 5
}

def apply_patch():
    for agent in r.smembers(AGENTS):
        name = agent.decode("utf-8")
        for key, val in PATCH.items():
            r.hset(f"agent:{name}:config", key, val)
        print(f"[patch] Patched {name}")

if __name__ == "__main__":
    apply_patch()



# inject_profiler.py
"""
Injects CPU/RAM profiling entries per agent into Redis.
Logged once per second for 15 seconds. Used for short-term load testing.
"""

import time
import redis
import random

r = redis.Redis()
AGENTS = [f"agent:{i}" for i in range(1, 6)]


def profile():
    for agent in AGENTS:
```

```
        cpu = round(random.uniform(1, 15), 2)
        ram = random.randint(32, 512)
        r.hset(f"{agent}:profile", mapping={
            "cpu_percent": cpu,
            "memory_kb": ram * 1024
        })
        print(f"[inject] {agent} CPU={cpu}% RAM={ram}MB")


if __name__ == "__main__":
    for _ in range(15):
        profile()
        time.sleep(1)
```

```
# run_logicshredder.py
"""
Main entrypoint for symbolic swarm boot.
Verifies preconditions and starts all autonomous agents.
"""

import subprocess
import redis

AGENTS = ["seed", "scanner", "validator", "dreamer"]
QUEUE = "swarm:init"
r = redis.Redis()

def preload():
    for a in AGENTS:
        r.lpush(QUEUE, a)
    print("[init] Seeded init queue.")

def boot():
    preload()
    subprocess.call("python async_swarm_launcher.py", shell=True)

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

```
# patch_agents_config.py
"""
Patches all known agents' configuration parameters in Redis.
Used to modify runtime priority, memory cap, or task mode.
"""

import redis
r = redis.Redis()

AGENTS = "swarm:agents"
PATCH = {
```

```
    "task_mode": "explore",
    "max_ram": 128,
    "priority": 5
}

def apply_patch():
    for agent in r.smembers(AGENTS):
        name = agent.decode("utf-8")
        for key, val in PATCH.items():
            r.hset(f"agent:{name}:config", key, val)
        print(f"[patch] Patched {name}")

if __name__ == "__main__":
    apply_patch()



# inject_profiler.py
"""
Injects CPU/RAM profiling entries per agent into Redis.
Logged once per second for 15 seconds. Used for short-term load testing.
"""

import time
import redis
import random

r = redis.Redis()
AGENTS = [f"agent:{i}" for i in range(1, 6)]


def profile():
    for agent in AGENTS:
        cpu = round(random.uniform(1, 15), 2)
        ram = random.randint(32, 512)
        r.hset(f"{agent}:profile", mapping={
            "cpu_percent": cpu,
            "memory_kb": ram * 1024
        })
        print(f"[inject] {agent} CPU={cpu}% RAM={ram}MB")


if __name__ == "__main__":
    for _ in range(15):
        profile()
        time.sleep(1)



# run_logicshredder.py
"""
Main entrypoint for symbolic swarm boot.
Verifies preconditions and starts all autonomous agents.
"""

import subprocess
```

```
import redis

AGENTS = ["seed", "scanner", "validator", "dreamer"]
QUEUE = "swarm:init"
r = redis.Redis()

def preload():
    for a in AGENTS:
        r.lpush(QUEUE, a)
    print("[init] Seeded init queue.")

def boot():
    preload()
    subprocess.call("python async_swarm_launcher.py", shell=True)

if __name__ == "__main__":
    boot()



# train_utils.py
"""
Shared helper functions for training symbolic-to-text models.
Includes fragment flattening, text normalization, and batching.
"""

import yaml
import os


def load_fragments(path):
    data = []
    for f in os.listdir(path):
        if f.endswith(".yaml"):
            with open(os.path.join(path, f), 'r') as file:
                frag = yaml.safe_load(file)
                text = f"{frag.get('claim')}\nEMOTION: {frag.get('emotion', {})}"
                data.append(text)
    return data


def batch_fragments(fragments, batch_size=4):
    return [fragments[i:i + batch_size] for i in range(0, len(fragments), batch_size)]


def normalize_text(s):
    return s.replace('\n', ' ').strip()


# data_utils.py
"""
Prepares raw symbolic data for tokenization and embedding steps.
Sorts, deduplicates, and filters fragment text lines.
"""
```

```
def deduplicate(data):
    seen = set()
    result = []
    for item in data:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result


def filter_short(lines, min_len=20):
    return [line for line in lines if len(line) >= min_len]


def sort_by_emotion(data, key='curiosity'):
    return sorted(data, key=lambda x: x.get('emotion', {}).get(key, 0), reverse=True)


# utils.py
"""
Assorted glue logic and JSON helpers shared across toolchain.
"""

import json

def read_json(path):
    with open(path, 'r') as f:
        return json.load(f)

def write_json(path, obj):
    with open(path, 'w') as f:
        json.dump(obj, f, indent=2)

def flatten_dict(d, parent_key='', sep='.'):
    items = []
    for k, v in d.items():
        new_key = f"{parent_key}{sep}{k}" if parent_key else k
        if isinstance(v, dict):
            items.extend(flatten_dict(v, new_key, sep=sep).items())
        else:
            items.append((new_key, v))
    return dict(items)




# fragment_loader.py
"""
Walks a directory of fragments and loads all YAML into memory
for symbolic inspection or batch processing.
"""

import os
```

```
import yaml


def load_all(path):
    frags = []
    for f in os.listdir(path):
        if f.endswith(".yaml"):
            with open(os.path.join(path, f), 'r') as file:
                frags.append(yaml.safe_load(file))
    return frags


# validator.py
"""
Validates symbolic fragments for required fields and basic consistency.
Warns on missing emotional structure or malformed claims.
"""

REQUIRED_FIELDS = ["id", "claim", "created", "emotion", "metadata"]


def validate_fragment(frag):
    missing = [f for f in REQUIRED_FIELDS if f not in frag]
    if missing:
        return False, f"Missing: {missing}"
    if not isinstance(frag.get("emotion"), dict):
        return False, "Emotion field is not a dictionary"
    return True, "OK"


# symbolic_explanation_probe.py
"""
Takes symbolic fragments and scores their clarity and emotional readability.
Used for explanation ranking and trust-level visualizations.
"""

def clarity_score(frag):
    claim = frag.get("claim", "")
    emo = frag.get("emotion", {})
    length = len(claim.split())
    weight = sum(emo.values())
    return round((length / 20) + weight, 2)


def explain(frag):
    return {
        "id": frag.get("id"),
        "summary": frag.get("claim", "[no claim]"),
        "clarity": clarity_score(frag)
    }


# neuro_lock.py
"""
```

```
Symbolic mutex for memory zones. Prevents conflicting agent writes
by claiming and releasing zones with temporary UUID locks.
"""

import redis
import uuid

r = redis.Redis()
LOCK_KEY = "neuro:lock"


def acquire():
    lock_id = str(uuid.uuid4())
    if r.setnx(LOCK_KEY, lock_id):
        return lock_id
    return None


def release(lock_id):
    if r.get(LOCK_KEY) == lock_id.encode():
        r.delete(LOCK_KEY)
        return True
    return False




# async_swarm_launcher.py
"""
Launches all symbolic agents asynchronously in subprocesses.
Used to boot a full swarm from a single controller call.
"""

import subprocess

AGENTS = [
    "seed_agent.py",
    "file_sage_agent.py",
    "token_agent.py",
    "dreamwalker.py",
    "meta_agent.py"
]

def launch():
    for agent in AGENTS:
        subprocess.Popen(["python", agent])
        print(f"[launcher] Spawned: {agent}")

if __name__ == "__main__":
    launch()


# adaptive_installer.py
"""
```

```
Installer script that checks for dependencies, creates folders,
and sets up symbolic swarm workspace for first-time install.
"""

import os
import subprocess

FOLDERS = [
    "fragments/core",
    "fragments/devoured",
    "logs",
    "configs",
    "maps",
    "shim/nvme_blocks"
]

def install():
    for folder in FOLDERS:
        os.makedirs(folder, exist_ok=True)
        print(f"[install] Created {folder}")
    subprocess.call(["pip", "install", "psutil", "pyyaml", "numpy", "redis"])

if __name__ == "__main__":
    install()



# cold_logic_mover.py
"""
Moves low-heat or decayed fragments to deep storage or archival directories.
Reduces memory pressure during swarm operation.
"""

import os
import shutil
import yaml
from datetime import datetime, timedelta

SRC = "fragments/core"
DEST = "fragments/archived"

def move_old():
    os.makedirs(DEST, exist_ok=True)
    for f in os.listdir(SRC):
        if f.endswith(".yaml"):
            path = os.path.join(SRC, f)
            with open(path, 'r') as file:
                frag = yaml.safe_load(file)
            created = frag.get("created")
            if not created:
                continue
            age = (datetime.utcnow() - datetime.fromisoformat(created)).days
            if age > 30:
                shutil.move(path, os.path.join(DEST, f))
                print(f"[mover] Cold-moved: {f}")
```

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


# start_logicshredder.bat
@echo off
python run_logicshredder.py


# start_logicshredder_silent.bat
@echo off
start /min python run_logicshredder.py



# auto_configurator.py
"""
Scans hardware and emits a baseline symbolic swarm config.
"""

import json, os, psutil

CONFIG_PATH = "configs/system_config.json"


def generate():
    profile = {
        "cpu": psutil.cpu_count(logical=True),
        "ram_mb": psutil.virtual_memory().total // (1024 * 1024),
        "disk_gb": psutil.disk_usage("/").free // (1024 * 1024 * 1024),
    }
    os.makedirs("configs", exist_ok=True)
    with open(CONFIG_PATH, 'w') as f:
        json.dump(profile, f, indent=2)
    print(f"[configurator] Wrote: {CONFIG_PATH}")


if __name__ == "__main__":
    generate()


# config_loader.py
"""
Loads any symbolic config JSON into memory for agent prep.
"""

import json

CONFIG_PATH = "configs/system_config.json"


def load_config():
    with open(CONFIG_PATH, 'r') as f:
```

```
        return json.load(f)


# config_access.py
"""
Exposes current config as a callable CLI helper.
"""

from config_loader import load_config

if __name__ == "__main__":
    config = load_config()
    for k, v in config.items():
        print(f"{k.upper()}: {v}")



# compile_to_pdf.py
"""
Combines all .py files in a folder into a single PDF document.
"""

from fpdf import FPDF
import os

class CodePDF(FPDF):
    def header(self):
        self.set_font("Courier", 'B', 10)
        self.cell(0, 10, "Symbolic AI Code Archive", ln=True, align='C')

    def add_code_file(self, path):
        self.set_font("Courier", size=8)
        self.add_page()
        self.multi_cell(0, 5, f"-- {path} --\n")
        with open(path, 'r', encoding='utf-8', errors='ignore') as f:
            for line in f:
                self.multi_cell(0, 5, line)


def main():
    pdf = CodePDF()
    for file in os.listdir("."):
        if file.endswith(".py") and file != __file__:
            pdf.add_code_file(file)
    pdf.output("compiled_code.pdf")


if __name__ == "__main__":
    main()


# constants.py
"""
Shared constants across the symbolic runtime.
"""
```

```
EMOTIONS = ["curiosity", "shame", "awe", "doubt", "hope"]
CORE_DIR = "fragments/core"
CONFIG_PATH = "configs/system_config.json"
DEFAULT_AGENT_LIMIT = 12
```

```
# benchmark_agent.py
"""
Evaluates an agent's throughput by measuring fragment processed/sec.
Used to compare agent efficiency on symbolic workloads.
"""

import time
import random

AGENT_NAME = "test_benchmark_agent"
FRAGMENT_COUNT = 500


def fake_process():
    time.sleep(random.uniform(0.001, 0.005))


def run_benchmark():
    start = time.time()
    for _ in range(FRAGMENT_COUNT):
        fake_process()
    duration = time.time() - start
    print(f"[benchmark] Agent '{AGENT_NAME}' processed {FRAGMENT_COUNT} fragments in {round(duration, 2)} sec")
    print(f"[benchmark] ? {round(FRAGMENT_COUNT / duration, 2)} frags/sec")


if __name__ == "__main__":
    run_benchmark()


# compare-llama-bench.py
"""
Dummy benchmark comparison tool that simulates running inference
against several language model backends.
"""

import time
import random

MODELS = ["GPT-4", "LLaMA-2", "SymbolNet-Beta"]
REQUESTS = 100


def simulate_inference():
    return random.uniform(0.05, 0.3)
```

```
def compare():
    for model in MODELS:
        total = 0.0
        for _ in range(REQUESTS):
            total += simulate_inference()
        avg = total / REQUESTS
        print(f"[{model}] avg latency: {round(avg * 1000, 2)} ms")

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

```
# bench.py
"""
Top-level orchestrator for symbolic benchmarks across agents.
Can be expanded to write reports or save logs.
"""

from benchmark_agent import run_benchmark

if __name__ == "__main__":
    print("=== Symbolic Benchmark Suite ===")
    run_benchmark()
```

```
# redis_publisher.py
"""
Simple Redis pub tool to broadcast symbolic messages to a channel.
"""

import redis
import time

r = redis.Redis()
channel = "symbolic:broadcast"

def publish_loop():
    while True:
        msg = input("> ")
        r.publish(channel, msg)
        print(f"[pub] sent: {msg}")

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

```
# redis_subscriber.py
"""
Redis subscriber to listen to symbolic swarm channels.
"""
```

```
import redis

r = redis.Redis()
pubsub = r.pubsub()
pubsub.subscribe("symbolic:broadcast")

print("[sub] listening...")

for message in pubsub.listen():
    if message['type'] == 'message':
        print(f"[recv] {message['data'].decode()}")



# install_everything_brainy.py
"""
Installs system requirements, Python packages, Redis, and symbolic agents.
"""

import os
import subprocess

print("[setup] Installing brain dependencies")
subprocess.call(["pip", "install", "redis", "psutil", "pyyaml", "numpy", "fpdf", "fastapi", "uvicorn"])

FOLDERS = ["fragments", "logs", "configs", "maps"]
for f in FOLDERS:
    os.makedirs(f, exist_ok=True)
    print(f"[setup] Created {f}")

print("[setup] All symbolic systems initialized")



# neurostore_backend_setup.py
"""
Bootstraps a symbolic FastAPI backend for NeuroStore tools.
"""

from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/status")
def get_status():
    return {"status": "alive", "agents": 4, "fragments": 112}

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)



# install_react_gui_prereqs.py
"""
Installs prerequisites for React GUI frontends tied to symbolic swarm control.
```

```
"""

import subprocess
import os

print("[react] Installing frontend dependencies...")
subprocess.call(["npm", "install", "--force"])
subprocess.call(["npm", "install", "axios", "vite", "react-router-dom"])

print("[react] Setup complete.")



# symbol_seed_generator.py
"""
Seeds symbolic YAML structures with basic ideas and emotional tags.
"""

import uuid, yaml
from datetime import datetime
import os

SEEDS = ["Contradiction fuels recursion.", "Belief is symbolic inertia."]
OUT_DIR = "fragments/core/seeds"

os.makedirs(OUT_DIR, exist_ok=True)

for idea in SEEDS:
    doc = {
        "id": str(uuid.uuid4()),
        "claim": idea,
        "created": datetime.utcnow().isoformat(),
        "emotion": {"curiosity": 0.6},
        "metadata": {"origin": "symbol_seed_generator"},
        "tags": ["seed"]
    }
    fname = os.path.join(OUT_DIR, f"seed_{doc['id']}.yaml")
    with open(fname, 'w') as f:
        yaml.dump(doc, f)
    print(f"[seed] {fname}")



# quant_prompt_feeder.py
"""
Extracts claim + emotion into quant-style symbolic prompts for training.
"""

import yaml, os
FRAGS = "fragments/core"

for f in os.listdir(FRAGS):
    if f.endswith(".yaml"):
        with open(os.path.join(FRAGS, f)) as y:
            d = yaml.safe_load(y)
```

```
        print(f"PROMPT: {d['claim']} [EMO: {d.get('emotion', {})}]")


# quant_feeder_setup.py
"""
Sets up directory and prints YAML prompt metadata preview.
"""

import os, yaml

os.makedirs("quant_prompts", exist_ok=True)

with open("quant_prompts/manifest.yaml", 'w') as m:
    yaml.dump({"generated": True, "count": 0}, m)

print("[quant] Prompt dir and manifest created")



# word_dict_gen.py
"""
Builds a word frequency dict from YAML fragments.
"""

import yaml, os
from collections import Counter

words = Counter()

for f in os.listdir("fragments/core"):
    if f.endswith(".yaml"):
        with open(os.path.join("fragments/core", f)) as y:
            d = yaml.safe_load(y)
        tokens = d.get("claim", "").lower().split()
        for word in tokens:
            words[word] += 1

print(words.most_common(10))



# requirements.py
"""
Dump pip dependencies for reproducible symbolic environment.
"""

REQUIREMENTS = [
    "redis", "numpy", "pyyaml", "psutil", "uvicorn", "fastapi", "fpdf"
]

with open("requirements.txt", 'w') as r:
    r.write("\n".join(REQUIREMENTS))

print("[reqs] Wrote requirements.txt")
```

```
# train_pararule.py
"""
Symbolic para-rule trainer (text ? logic-style label pairs).
"""

from utils_pararule import load_dataset, train_model

if __name__ == '__main__':
    X, y = load_dataset("data/pararule.tsv")
    model = train_model(X, y)
    print("[train] done")


# utils_pararule.py
"""
Pararule dataset loader and symbolic classifier wrapper.
"""

import csv
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer


def load_dataset(path):
    with open(path) as f:
        rows = list(csv.reader(f, delimiter='\t'))
    return [r[0] for r in rows], [r[1] for r in rows]


def train_model(X, y):
    vec = CountVectorizer()
    Xv = vec.fit_transform(X)
    clf = LogisticRegression()
    clf.fit(Xv, y)
    return clf


# utils_conceptrule.py
"""
Functions for concept rule formatting and rule logic expansion.
"""


def encode_rule(subject, relation, object):
    return f"If {subject} has {relation}, then it also relates to {object}."


def batch_encode(triples):
    return [encode_rule(s, r, o) for s, r, o in triples]


# utils_conceptrule_csv.py
"""
```

```
Loads concept rule triples from CSV.
"""

import csv

def load_concept_csv(path):
    with open(path) as f:
        return [tuple(row) for row in csv.reader(f)]



# axioms_and_interfaces.py

"""
This file captures foundational symbolic axioms and helper interfaces
extracted from deep chat context and architectural notes.
These are core to the system?s philosophical design and emotional structure.
"""

# === Axiom: Stillness of Reference ===
"""
Still Point Axiom:
"Something must stay still so everything else can move."
Used as a symbolic anchor during high contradiction recursion cycles.
Tags: ["reference", "root", "inertia"]
"""

STILL_POINT_FRAGMENT = {
    "id": "00000000-stillpoint",
    "claim": "Something must stay still so everything else can move.",
    "created": "0000-00-00T00:00:00Z",
    "emotion": {"awe": 0.8},
    "metadata": {"origin": "axiom_seed"},
    "tags": ["axiom", "stillness"]
}


# === Emotional Decay Interface ===
"""
Wraps decay engine and exposes function for symbolic agents to
request emotion-reduction over time or by contradiction events.
"""

import os
import yaml
from datetime import datetime

FRAGMENTS_DIR = "fragments/core"

def decay_by_request(fragment_id):
    frag_path = os.path.join(FRAGMENTS_DIR, fragment_id)
    if not os.path.exists(frag_path):
        print(f"[decay] Missing fragment: {fragment_id}")
        return
```

```
    with open(frag_path, 'r') as f:
        frag = yaml.safe_load(f)

    emo = frag.get("emotion", {})
    for key in emo:
        old_val = emo[key]
        emo[key] = round(emo[key] * 0.9, 2)  # 10% decay
        print(f"[decay] {key}: {old_val} ? {emo[key]}")

    frag["emotion"] = emo
    frag["metadata"]["decayed"] = datetime.utcnow().isoformat()

    with open(frag_path, 'w') as f:
        yaml.dump(frag, f)
    print(f"[decay] Fragment {fragment_id} updated.")


# === Usage ===
"""
from axioms_and_interfaces import STILL_POINT_FRAGMENT, decay_by_request
"""



# neuro_auditor.py

"""
NeuroAuditor ? extracted from 'Monday - Blazed VM Architecture.html'
Provides runtime scanning of fragments, identifying missing metadata,
emotion gaps, contradiction overload, or decay-state inconsistencies.

This system can be hooked into agent logic or run as an independent daemon.
"""

import os
import yaml

FRAG_DIR = "fragments/core"
REQUIRED_KEYS = ["claim", "created", "emotion", "metadata"]


def scan_fragments():
    issues = []
    for fname in os.listdir(FRAG_DIR):
        if fname.endswith(".yaml"):
            path = os.path.join(FRAG_DIR, fname)
            with open(path, 'r') as f:
                try:
                    frag = yaml.safe_load(f)
                except Exception as e:
                    issues.append((fname, f"Parse error: {e}"))
                    continue
```

```
            for key in REQUIRED_KEYS:
                if key not in frag:
                    issues.append((fname, f"Missing key: {key}"))

            emo = frag.get("emotion", {})
            if not emo or not isinstance(emo, dict):
                issues.append((fname, "Invalid or missing emotion block"))

            if "decayed_at" in frag.get("metadata", {}):
                if not frag.get("tags") or "decayed" not in frag["tags"]:
                    issues.append((fname, "Marked decayed but missing 'decayed' tag"))

    return issues


def report():
    results = scan_fragments()
    if not results:
        print("[audit] All fragments pass.")
        return
    for fname, issue in results:
        print(f"[audit] {fname}: {issue}")


if __name__ == "__main__":
    report()




# symbolic_concepts_unwritten.py

"""
This file contains high-level design fragments, conceptual interfaces,
and pseudocode extracted from unreleased architecture notes.
Intended for future implementation in the symbolic swarm ecosystem.
"""

# === emotion_tune.py (stub) ===
"""
Adjusts the emotional weight block in any YAML fragment.
May be used by agents, CLI, or user override interface.
"""

def tune_emotion(fragment_path, multiplier):
    """ Scales all emotion weights by multiplier (e.g. 0.85) """
    pass  # TODO: implement


# === lineage_mapper.py (stub) ===
"""
Generates belief ancestry maps by tracing 'origin' ? 'derived_from' links.
Intended to output .dot graph or JSON lineage tree.
"""
```

```
def map_lineage(directory):
    """ Traverse YAMLs and cluster by shared ancestry """
    pass  # TODO: implement


# === Symbolic Camouflage ===
"""
Fragments are tagged with `camouflage:true` and ignored until a triggering
phrase, emotion, or contradiction state activates them. This simulates repression.
"""

# Example Trigger Code:
# if frag.get("metadata", {}).get("camouflage") and matches(trigger_pattern):
#     unhide and evaluate fragment


# === Mutation Energy Awareness ===
"""
Track mutation rate and link symbolic CPU load to emotional fatigue.
"""

# Pseudocode:
# if mutation_count > threshold:
#     agent.emotion["fatigue"] += 0.2
#     agent.performance -= 10%


# === Supernode Beliefs ===
"""
Fragments that emerge repeatedly across the swarm get clustered
into supernodes ? abstract meta-beliefs that influence emotional routing.
"""

# e.g.:
# if same claim appears in >5 agents:
#     promote to supernode, increase routing priority


# === Curiosity Engine ===
"""
Scores uncertainty, rarity, and contradiction density to direct agent attention.
Can be used to fuel Dreamwalker, seed generator, and decay inversions.
"""

# e.g.:
# score = 0.6 * unknown_refs + 0.4 * rare_tags + 0.2 * contradictions
# if score > 1.2:
#     trigger dreamwalker event


# === Alignment Layer ===
"""
A human-readable layer that overrides swarm contradictions
```