```
==== agent_log_viewer.py ====
import json
from pathlib import Path

LOG_DIR = "logs"


def view_logs(agent=None, contains=None):
    print("\n? Log Viewer Results:")
    for file in Path(LOG_DIR).glob("*.jsonl"):
        with open(file, "r", encoding="utf-8") as f:
            for line in f:
                try:
                    entry = json.loads(line)
                    if agent and entry.get("role") != agent:
                        continue
                    if contains and contains.lower() not in json.dumps(entry).lower():
                        continue
                    print(f"[{entry.get('timestamp')}] {entry.get('role')} ? {entry.get('task', '')[:60]}")
                except:
                    continue

if __name__ == "__main__":
    import sys
    role = sys.argv[1] if len(sys.argv) > 1 else None
    keyword = sys.argv[2] if len(sys.argv) > 2 else None
    view_logs(agent=role, contains=keyword)

==== agentiq_profiler_agent.py ====
import uuid
import json
from datetime import datetime
from tools.parallel_rag_query import query_all
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.run_model import run_model

MODEL = "phi:2"
ROLE = "profiler"

def profile_fragment_flow(task):
    context = query_all(task)
    prompt = build_prompt_from_ram(task, context)
    response = run_model(MODEL, prompt)
    trace = {
        "task": task,
        "agent": ROLE,
        "model": MODEL,
        "trace_id": str(uuid.uuid4())[:8],
        "timestamp": datetime.utcnow().isoformat(),
        "input": prompt,
        "output": response,
        "token_cost_est": len(prompt.split()) + len(response.split())
    }
```

```
    with open("logs/agentiq_traces.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps(trace) + "\n")
    return response

if __name__ == "__main__":
    import sys
    task = " ".join(sys.argv[1:]) or "profile summarizer usage over last 50 tasks"
    profile_fragment_flow(task)

==== auto_writer.py ====
import uuid
import json
import asyncio
from tools.parallel_rag_query import query_all
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.run_model import run_model
from datetime import datetime

MODEL = "mistral:7b"
ROLE = "writer"

async def generate_report(task: str):
    context = query_all(task)
    prompt = build_prompt_from_ram(task, context)
    result = await run_model(MODEL, prompt)
    log(task, result)
    return result

def log(query, result):
    doc_id = str(uuid.uuid4())[:8]
    timestamp = datetime.utcnow().isoformat()
    entry = {
        "id": doc_id,
        "role": ROLE,
        "timestamp": timestamp,
        "task": query,
        "output": result
    }
    with open(f"logs/generated_reports.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps(entry) + "\n")

if __name__ == "__main__":
    import sys
    task = " ".join(sys.argv[1:]) or "write a summary of recent memory events"
    asyncio.run(generate_report(task))

==== compile_to_pdf.py ====
import os
from pathlib import Path
from fpdf import FPDF

# Extensions to include
FILE_EXTENSIONS = [".py", ".yaml", ".yml", ".json", ".txt"]
```

```
class CodePDF(FPDF):
    def __init__(self):
        super().__init__()
        self.set_auto_page_break(auto=True, margin=15)
        self.add_page()
        self.set_font("Courier", size=8)

    def add_code_file(self, filepath):
        self.set_font("Courier", size=8)
        self.multi_cell(0, 5, f"\n==== {filepath} ====\n")
        try:
            with open(filepath, 'r', encoding='utf-8', errors='ignore') as f:
                for line in f:
                    clean_line = ''.join(c if 0x20 <= ord(c) <= 0x7E or c in '\t\n\r' else '?' for c in line)
                    self.multi_cell(0, 5, clean_line.rstrip())
        except Exception as e:
            self.multi_cell(0, 5, f"[Error reading {filepath}: {e}]\n")

def gather_files(root_dir, extensions):
    return [
        f for f in Path(root_dir).rglob("*")
         if f.is_file() and f.suffix.lower() in extensions and "venv" not in f.parts and "__pycache__" not in
f.parts
    ]

def main(root=".", output="symbolic_manifesto.pdf"):
    pdf = CodePDF()
    files = gather_files(root, FILE_EXTENSIONS)

    if not files:
        print("[!] No matching files found.")
        return

    for file in sorted(files):
        pdf.add_code_file(file)

    pdf.output(output)
    print(f"[?] Compiled {len(files)} files into: {output}")

if __name__ == "__main__":
    main()

==== create_memory_dbs.py ====
from pathlib import Path

base_db = Path("C:/real_memory_system/memory.duckdb")
target_dir = Path("C:/real_memory_system/memory_db")
target_dir.mkdir(parents=True, exist_ok=True)

categories = [
    "hardware", "logs", "scripts", "errors", "summaries",
    "projects", "queries", "training", "planning", "reflections"
]
```

```
for name in categories:
    dest = target_dir / f"{name}.duckdb"
    if not dest.exists():
        with open(base_db, "rb") as src, open(dest, "wb") as out:
            out.write(src.read())
        print(f"[?] Created: {dest}")
    else:
        print(f"[?] Already exists: {dest}")

==== fragment_feedback_loop.py ====
import os
import json
import duckdb
from pathlib import Path
from datetime import datetime

MEMORY_DIR = "C:/real_memory_system"
LOG_FILE = "logs/report_writer_output.jsonl"
CATEGORY = "auto_ingest"
SUBCATEGORY = "writer_feedback"


def insert_fragment(category, fragment):
    db_path = os.path.join(MEMORY_DIR, f"{category}.duckdb")
    if not os.path.exists(db_path):
        print(f"[SKIP] No DB found for category '{category}'")
        return

    conn = duckdb.connect(db_path)
    conn.execute("""
        INSERT INTO fragments (
            id, claim, sub_category, confidence, tags, timestamp, filepath, content
        ) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """, (
        fragment["id"],
        fragment["claim"],
        SUBCATEGORY,
        0.95,
        ["feedback", "agent", "auto"],
        fragment["timestamp"],
        None,
        fragment["output"]
    ))
    conn.close()
    print(f"[?] Ingested fragment {fragment['id']} into [{category}]")


def feedback_to_memory():
    with open(LOG_FILE, "r", encoding="utf-8") as f:
        for line in f:
            frag = json.loads(line)
            frag["claim"] = frag["task"][:100]
            insert_fragment(CATEGORY, frag)
```

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

==== goal_router.py ====
# goal_router.py (Windows-Only Version ? Table Debug Mode, Schema Fixed)

import duckdb
import os
from concurrent.futures import ThreadPoolExecutor, as_completed

# Windows-native path
REAL_ROOT = r"C:\real_memory_system"
DB_DIR = REAL_ROOT  # memory.duckdb is directly inside this dir

LAYERED_CATEGORIES = [
    "fragments"  # actual table inside memory.duckdb
]

def list_tables(db_path):
    try:
        con = duckdb.connect(db_path)
        tables = con.execute("SHOW TABLES").fetchall()
        con.close()
        print(f"\n? Tables in {db_path}:")
        for table in tables:
            print(f" - {table[0]}")
    except Exception as e:
        print(f"Error checking tables in {db_path}: {e}")

def search_category(category, keyword, limit):
    db_path = os.path.join(DB_DIR, "memory.duckdb")
    results = []
    if not os.path.exists(db_path):
        return results
    list_tables(db_path)
    try:
        con = duckdb.connect(db_path)
        query = f"""
            SELECT filepath, content FROM {category}
            WHERE content ILIKE '%{keyword}%'
            LIMIT {limit}
        """
        matches = con.execute(query).fetchall()
        con.close()
        for match in matches:
            results.append({
                "category": category,
                "filepath": match[0],
                "filename": os.path.basename(match[0]),
                "snippet": match[1]
            })
    except Exception as e:
        print(f"[!] Error reading {db_path}: {e}")
    return results
```

```
def search_memory(keyword, limit=10):
    results = []
    with ThreadPoolExecutor(max_workers=8) as executor:
        futures = {
            executor.submit(search_category, category, keyword, limit): category
            for category in LAYERED_CATEGORIES
        }
        for future in as_completed(futures):
            try:
                results.extend(future.result())
            except Exception as e:
                print(f"[!] Thread error: {e}")
    return results

if __name__ == "__main__":
    import sys
    if len(sys.argv) < 2:
        print("Usage: python goal_router.py <keyword> [limit]")
        exit(1)
    keyword = sys.argv[1]
    limit = int(sys.argv[2]) if len(sys.argv) > 2 else 10
    results = search_memory(keyword, limit)
    print(f"\n? Found {len(results)} results for '{keyword}':\n")
    for r in results:
        print(f"[{r['category']}] {r['filename']}\n  {r['snippet'][:200].strip()}\n  ? {r['filepath']}\n")

==== intent_router_agent.py ====
import json
import uuid
from datetime import datetime
from tools.run_model import run_model

MODEL = "phi:2"
ROLE = "intent_router"
INPUT_FILE = "tasks/raw_input_queue.json"
OUTPUT_FILE = "tasks/agent_queue.json"


def classify(prompt):
    classification_prompt = f"""
Classify the following task into one of the roles: summarizer, coder, retriever, reasoner, generalist.
Task: {prompt}
Respond with only the role.
"""
    result = run_model(MODEL, classification_prompt)
    return result.strip().lower()


def route_tasks():
    with open(INPUT_FILE, "r", encoding="utf-8") as f:
        raw = json.load(f)

    routed = {}
```

```
    for tid, prompt in raw.items():
        role = classify(prompt)
        routed[role] = prompt
        log_decision(tid, role, prompt)

    with open(OUTPUT_FILE, "w", encoding="utf-8") as f:
        json.dump(routed, f, indent=2)

    print(f"? Routed {len(routed)} tasks to roles.")


def log_decision(tid, role, prompt):
    with open("logs/intent_router_log.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps({
            "id": tid,
            "timestamp": datetime.utcnow().isoformat(),
            "role": role,
            "task": prompt
        }) + "\n")

if __name__ == "__main__":
    route_tasks()

==== llm_benchmark_runner.py ====
import subprocess
import time
import sys

MODELS = [
    "phi:2",
    "tinyllama:1.1b-chat",
    "tinydolphin",
    "zephyr:1.1b",
    "openhermes:1.5-mistral",
    "dolphin-phi:2",
    "mistral:instruct"
]

TASK = "Summarize the function and purpose of the router_controller.py file in a single paragraph."

print("\n[? LLM BENCHMARK STARTED]\n")

for model in MODELS:
    print(f"?? Testing model: {model}")
    start = time.time()
    try:
        result = subprocess.run(
            ["ollama", "run", model],
            input=TASK.encode("utf-8"),
            capture_output=True,
            timeout=60
        )
        duration = time.time() - start
        print(f"? Time: {duration:.2f}s")
```

```
        print("? Output:", result.stdout.decode("utf-8").strip().splitlines()[0])
    except Exception as e:
        print("? Error:", str(e))
    print("\n" + ("-" * 60) + "\n")

==== llm_ram_prompt_builder.py ====
# llm_ram_prompt_builder.py
import json
from pathlib import Path

RAM_CACHE = Path("C:/real_memory_system/cache/ram_fragments.json")

def build_prompt_from_ram(task, token_limit=16000):
    with open(RAM_CACHE, "r", encoding="utf-8") as f:
        frags = json.load(f)

    output = []
    total_chars = 0
    for f in frags:
        chunk = f"# {f['timestamp']} [{', '.join(f['tags'])}] ({f['sub']})\n{f['claim']}\n{f['content']}\n"
        total_chars += len(chunk)
        if total_chars > token_limit * 4:
            break
        output.append(chunk)

    prompt = "\n\n".join(output)
    prompt += f"\n\n### TASK\n{task}"
    return prompt

if __name__ == "__main__":
    test = build_prompt_from_ram("Improve nested VM fault tolerance")
    print(test[:2000])

==== media_fragment_indexer.py ====
import os
import pytesseract
from PIL import Image
from pathlib import Path
import uuid
import json
from datetime import datetime

INPUT_DIR = "C:/real_memory_system/FEEDING_TIME"
OUTPUT_LOG = "logs/media_index_log.jsonl"
CATEGORY = "media"
SUBCATEGORY = "ocr"


def process_image(filepath):
    try:
        img = Image.open(filepath)
        text = pytesseract.image_to_string(img)
        if len(text.strip()) == 0:
            return None
```

```
        return text.strip()
    except Exception as e:
        print(f"[SKIP] {filepath}: {e}")
        return None


def index_images():
    for file in Path(INPUT_DIR).rglob("*.png"):
        extract_and_log(file)
    for file in Path(INPUT_DIR).rglob("*.jpg"):
        extract_and_log(file)
    for file in Path(INPUT_DIR).rglob("*.jpeg"):
        extract_and_log(file)


def extract_and_log(path):
    text = process_image(path)
    if not text:
        return

    fragment = {
        "id": str(uuid.uuid4())[:8],
        "claim": text[:100],
        "sub_category": SUBCATEGORY,
        "confidence": 0.9,
        "tags": ["ocr", "image", "indexed"],
        "timestamp": datetime.utcnow().isoformat(),
        "filepath": str(path),
        "content": text
    }

    with open(OUTPUT_LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(fragment) + "\n")

    print(f"[?] Indexed {path.name}")


if __name__ == "__main__":
    index_images()

==== media_to_memory.py ====
import json
import uuid
import duckdb
from datetime import datetime
from pathlib import Path

LOG_PATH = "logs/media_index_log.jsonl"
DB_PATH = "C:/real_memory_system/media.duckdb"


def ingest():
    if not Path(LOG_PATH).exists():
        print("?? No OCR logs found.")
```

```
        return

    conn = duckdb.connect(DB_PATH)
    with open(LOG_PATH, "r", encoding="utf-8") as f:
        for line in f:
            try:
                data = json.loads(line)
                conn.execute("""
                    INSERT INTO fragments (
                        id, claim, sub_category, confidence, tags, timestamp, filepath, content
                    ) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
                """, (
                    data["id"],
                    data["claim"],
                    data["sub_category"],
                    0.85,
                    ["ocr", "image"],
                    data["timestamp"],
                    data["filepath"],
                    data["content"]
                ))
                print(f"[?] Injected {data['id']} into memory DB")
            except Exception as e:
                print(f"[SKIP] Error: {e}")
    conn.close()

if __name__ == "__main__":
    ingest()

==== memory_log_viewer.py ====
# memory_log_viewer.py
import duckdb
from datetime import datetime

DB_PATH = "C:/real_memory_system/memory.duckdb"

def view_log(tag=None, sub_category=None, keyword=None, limit=50):
    conn = duckdb.connect(DB_PATH)

    base = "SELECT timestamp, claim, sub_category, tags, filepath FROM fragments WHERE 1=1"
    if tag:
        base += f" AND array_contains(tags, '{tag}')"
    if sub_category:
        base += f" AND sub_category = '{sub_category}'"
    if keyword:
        base += f" AND (claim ILIKE '%{keyword}%' OR filepath ILIKE '%{keyword}%')"

    base += " ORDER BY timestamp DESC LIMIT ?"
    rows = conn.execute(base, (limit,)).fetchall()
    conn.close()

    for ts, claim, sub, tags, path in rows:
        print(f"[{ts}] [{sub}] {claim} ({', '.join(tags)}) ? {path}")
```

```
if __name__ == "__main__":
    view_log(tag=None, sub_category=None, keyword=None)

==== memory_stack_boot.py ====
# memory_stack_boot.py
import subprocess
from query_context import get_context
from prompt_assembler import build_prompt
from datetime import datetime
import os

def run_sorter():
    print("[BOOT] Running Super Sorter...")
    subprocess.run(["python", "super_sorter_v6.py"])

def start():
    run_sorter()

    print("[BOOT] Memory ingestion complete.\n")
    category = input("? Category DB (e.g., scripts, notes): ").strip()
    tags = input("?? Tags (comma separated): ").strip().split(",")
    sub = input("? Sub-category (or blank): ").strip() or None
    keywords = input("? Keywords (optional): ").strip().split() or None
    instruction = input("? Task Instruction for LLM: ").strip()

    print("\n[QUERY] Building memory context...\n")
    memory_block = get_context(category, tags, sub, keywords)
    full_prompt = build_prompt(memory_block, instruction)

    with open("logs/session_history.log", "a", encoding="utf-8") as log:
        log.write(f"\n--- {datetime.now().isoformat()} ---\n")
        log.write(full_prompt + "\n")

    print("[? FINAL PROMPT]\n")
    print(full_prompt)

    print("\n[? Sending to Ollama...]\n")
    subprocess.run(["ollama", "run", "mistral"], input=full_prompt.encode("utf-8"))

if __name__ == "__main__":
    start()

==== memory_tools_config.json ====
[
  {
    "name": "search_memory",
    "description": "Search indexed memory fragments for any keyword, returning content and file locations.",
    "parameters": {
      "type": "object",
      "properties": {
        "keyword": {
          "type": "string",
          "description": "The keyword or phrase to search for in memory."
        },
```

```
        "limit": {
          "type": "integer",
          "description": "Max number of memory results to return.",
          "default": 10
        }
      },
      "required": ["keyword"]
    },
    "run": "python C:/real_memory_system/tools/goal_router.py {keyword} {limit}"
  }
]

==== micro_agent_template.py ====
import os
import json
import uuid
import asyncio
from tools.parallel_rag_query import query_all
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.run_model import run_model

AGENT_ID = os.getenv("AGENT_ID", str(uuid.uuid4())[:8])
AGENT_ROLE = os.getenv("AGENT_ROLE", "summarizer")
AGENT_MODEL = os.getenv("AGENT_MODEL", "phi:2")

async def micro_agent_loop():
    while True:
        user_input = await get_task()
        context = query_all(user_input)
        prompt = build_prompt_from_ram(user_input, context)
        result = await run_model(AGENT_MODEL, prompt)
        log_result(user_input, result)
        await asyncio.sleep(0.1)

def log_result(query, result):
    with open(f"logs/{AGENT_ID}_log.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps({
            "agent": AGENT_ROLE,
            "model": AGENT_MODEL,
            "query": query,
            "output": result,
            "uuid": AGENT_ID
        }) + "\n")

async def get_task():
    with open("tasks/agent_queue.json", "r", encoding="utf-8") as f:
        data = json.load(f)
    return data.get(AGENT_ROLE, "summarize: explain this stream")

if __name__ == "__main__":
    asyncio.run(micro_agent_loop())

==== multi_brain_dispatcher.py ====
# multi_brain_dispatcher.py
```

```
import asyncio
from llm_ram_prompt_builder import build_prompt_from_ram
import subprocess

MODEL_LIST = [
    ("mistral", "agent-refactor"),
    ("mistral", "log-analyzer"),
    ("codellama:7b", "config-auditor")
]

def run_model(model, task):
    prompt = build_prompt_from_ram(task)
    proc = subprocess.run(["ollama", "run", model], input=prompt.encode("utf-8"), capture_output=True)
    return (model, task, proc.stdout.decode("utf-8"))

async def run_all():
    loop = asyncio.get_event_loop()
    tasks = [
        loop.run_in_executor(None, run_model, model, task)
        for model, task in MODEL_LIST
    ]
    results = await asyncio.gather(*tasks)
    for r in results:
        print(f"\n=== [{r[0]}] {r[1]} ===\n{r[2][:2000]}")

if __name__ == "__main__":
    asyncio.run(run_all())

==== multi_model_broker.py ====
import os
import random
from tools.multi_model_broker import broker_task

# Config: roles mapped to preferred models, fallback list included
MODEL_PREFS = {
    "summarizer": ["phi:2", "mistral:7b"],
    "retriever": ["phi:2"],
    "coder": ["tinyllama:1.1b", "codellama:13b"],
    "planner": ["mistral:7b", "zephyr:7b"],
    "reasoner": ["zephyr:7b", "mistral:7b"],
    "fallback": ["mistral:7b"]
}

# Main function to route and execute
def broker_task(role, prompt, temperature=0.4):
    models = MODEL_PREFS.get(role, MODEL_PREFS["fallback"])
    for model in models:
        try:
            print(f"[BROKER] Trying model: {model}")
            output = run_model(model, prompt, temperature=temperature)
            return output
        except Exception as e:
            print(f"[FAIL] Model {model}: {e}")
            continue
```

```
    raise RuntimeError("No models available for task.")

if __name__ == "__main__":
    import sys
    role = sys.argv[1] if len(sys.argv) > 1 else "summarizer"
    task = " ".join(sys.argv[2:]) or "Summarize current system behavior"
    out = broker_task(role, task)
    print(f"\n? Final output:\n{out}")

==== parallel_rag_query.py ====
from pathlib import Path
import duckdb
import json

FRAGMENTS = Path("C:/real_memory_system/fragments")
AGENT_LOG = FRAGMENTS / "answers.jsonl"
AGENT_QUEUE = Path("C:/real_memory_system/tools/tasks/agent_queue.json")
OCR_DIR = FRAGMENTS / "media/ocr"


def query_sources(query, source_type):
    if source_type == "sql":
        return query_sql_memory(query)
    elif source_type == "unstructured":
        return query_unstructured_chunks(query)
    elif source_type == "agent":
        return query_agent_outputs(query)
    elif source_type == "log":
        return query_logs(query)
    elif source_type == "ocr":
        return query_ocr_fragments(query)
    return []


def query_sql_memory(query):
    conn = duckdb.connect(str(FRAGMENTS / "cat_memory.duckdb"))
    conn.execute("SET memory_limit='4GB'")
      rows = conn.execute("SELECT content FROM fragments WHERE content ILIKE ? LIMIT 10", ('%' + query +
'%',)).fetchall()
    return [r[0] for r in rows]


def query_unstructured_chunks(query):
    results = []
    for file in (FRAGMENTS / "text").glob("*.txt"):
        if query.lower() in file.read_text(encoding="utf-8").lower():
            results.append(file.read_text(encoding="utf-8"))
    return results


def query_agent_outputs(query):
    if AGENT_QUEUE.exists():
        data = json.loads(AGENT_QUEUE.read_text())
        return [json.dumps(entry) for entry in data if query.lower() in json.dumps(entry).lower()]
```

```
    return []


def query_logs(query):
    results = []
    if AGENT_LOG.exists():
        with AGENT_LOG.open("r", encoding="utf-8") as f:
            for line in f:
                if query.lower() in line.lower():
                    results.append(line.strip())
    return results


def query_ocr_fragments(query):
    results = []
    for file in OCR_DIR.glob("*.txt"):
        if query.lower() in file.read_text(encoding="utf-8").lower():
            results.append(file.read_text(encoding="utf-8"))
    return results

==== pdf_sql_retriever_agent.py ====
import os
import json
from tools.parallel_rag_query import query_all
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.run_model import run_model

MODEL = "zephyr:7b"
ROLE = "pdf_sql_retriever"

STRUCTURED_KEYS = ["customer", "order", "invoice", "date", "sql", "database"]
UNSTRUCTURED_HINTS = ["manual", "faq", "guide", "pdf", "document", "how do i"]


def classify_task(task):
    lower = task.lower()
    if any(k in lower for k in STRUCTURED_KEYS):
        return "structured"
    if any(k in lower for k in UNSTRUCTURED_HINTS):
        return "unstructured"
    return "hybrid"


def build_query(task):
    task_type = classify_task(task)
    context = query_all(task)
    prompt = build_prompt_from_ram(task, context)
    tagged = f"[{task_type.upper()}] {prompt}"
    return tagged


def run(task):
    query = build_query(task)
    result = run_model(MODEL, query)
```

```
    log(task, result)
    return result


def log(task, result):
    with open("logs/pdf_sql_retriever.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps({"task": task, "output": result, "type": classify_task(task)}) + "\n")



if __name__ == "__main__":
    import sys
    task = " ".join(sys.argv[1:]) or "How do I refund an order from invoice #12013?"
    print(run(task))

==== prompt_assembler.py ====
# prompt_assembler.py
from datetime import datetime

def build_prompt(memory_block: str, instruction: str = None):
    header = "### MEMORY CONTEXT\n"
    footer = "\n### TASK\n"
    base_instruction = instruction or "Use the memory above to generate relevant code, data, or analysis."

    return f"{header}{memory_block.strip()}{footer}{base_instruction.strip()}"

==== query_context.py ====
# query_context.py
import duckdb
from pathlib import Path
from datetime import datetime

DB_DIR = Path("C:/real_memory_system")
DEFAULT_LIMIT = 2048

def get_context(category, tags=None, sub=None, keywords=None, max_chars=DEFAULT_LIMIT):
    db_path = DB_DIR / f"{category}.duckdb"
    if not db_path.exists():
        raise FileNotFoundError(f"Missing DB: {db_path}")

    conn = duckdb.connect(str(db_path))
    base_query = "SELECT claim, content, tags, timestamp FROM fragments WHERE 1=1"
    clauses = []

    if sub:
        clauses.append(f"sub_category = '{sub}'")
    if tags:
        for tag in tags:
            clauses.append(f"array_contains(tags, '{tag}')")
    if keywords:
        for kw in keywords:
            clauses.append(f"(claim ILIKE '%{kw}%' OR content ILIKE '%{kw}%')")

    if clauses:
        base_query += " AND " + " AND ".join(clauses)
```

```
    base_query += " ORDER BY timestamp DESC"

    results = conn.execute(base_query).fetchall()
    context = []
    total_chars = 0

    for claim, content, tags, timestamp in results:
        block = f"# {timestamp} [{', '.join(tags)}]\n{claim}\n\n{content}\n"
        if total_chars + len(block) > max_chars:
            break
        context.append(block)
        total_chars += len(block)

    conn.close()
    return "\n".join(context)

==== rag_agent_scheduler.py ====
import json
import random
import asyncio
from pathlib import Path

AGENTS = json.loads(Path("tiny_model_registry.json").read_text())
SUBCATEGORIES = [
    "boot", "hardware", "scripts", "errors", "summaries",
    "queries", "training", "results", "reflections", "projects"
]

# Create a task queue from RAG categories
def generate_tasks():
    tasks = {}
    for agent in AGENTS:
        category = random.choice(SUBCATEGORIES)
        tasks[agent["role"]] = f"analyze subcategory: {category}"
    Path("tasks/agent_queue.json").write_text(json.dumps(tasks, indent=2))
    print("? Task queue assigned to agents.")

# Dispatch every N seconds
def scheduler_loop():
    while True:
        generate_tasks()
        asyncio.run(asyncio.sleep(15))

if __name__ == "__main__":
    scheduler_loop()

==== rag_to_ollama.py ====
# rag_to_ollama.py
import argparse
from query_context import get_context
from prompt_assembler import build_prompt
import subprocess
from datetime import datetime
```

```
def main():
    parser = argparse.ArgumentParser(description="Query memory DB and send prompt to Ollama")
    parser.add_argument("--category", required=True, help="DuckDB category (e.g. scripts)")
    parser.add_argument("--tags", default="", help="Comma-separated tags (e.g. vm,boot)")
    parser.add_argument("--sub", default="", help="Sub-category filter")
    parser.add_argument("--keywords", default="", help="Search keywords")
    parser.add_argument("--instruction", required=True, help="Instruction for LLM")

    args = parser.parse_args()
    tags = [t.strip() for t in args.tags.split(",") if t.strip()]
    keywords = [k.strip() for k in args.keywords.split(",") if k.strip()]
    sub = args.sub.strip() or None

    print("? Querying memory...")
    context = get_context(args.category, tags, sub, keywords)

    if not context.strip():
        print("?? No fragments found.")
        return

    prompt = build_prompt(context, args.instruction)
    timestamp = datetime.now().isoformat()

    with open("logs/session_history.log", "a", encoding="utf-8") as log:
        log.write(f"\n--- {timestamp} ---\n{prompt}\n")

    print("\n? Final Prompt:\n")
    print(prompt)

    print("\n? Sending to Ollama...\n")
    subprocess.run(["ollama", "run", "mistral"], input=prompt.encode("utf-8"))

if __name__ == "__main__":
    main()

==== ram_loader.py ====
# ram_loader.py
import duckdb
import json
from pathlib import Path

DB_PATH = "C:/real_memory_system/memory.duckdb"
RAM_CACHE_PATH = Path("C:/real_memory_system/cache/ram_fragments.json")
RAM_CACHE_PATH.parent.mkdir(parents=True, exist_ok=True)

def load_top_fragments(tags=None, sub=None, keywords=None, limit=10000):
    conn = duckdb.connect(DB_PATH)
    conn.execute("PRAGMA enable_external_access=true")

    base = "SELECT id, claim, tags, sub_category, timestamp, content FROM fragments WHERE 1=1"
    clauses = []

    if tags:
```

```
        for tag in tags:
            clauses.append(f"array_contains(tags, '{tag}')")
    if sub:
        clauses.append(f"sub_category = '{sub}'")
    if keywords:
        for kw in keywords:
            clauses.append(f"(claim ILIKE '%{kw}%' OR content ILIKE '%{kw}%')")

    if clauses:
        base += " AND " + " AND ".join(clauses)

    base += f" ORDER BY timestamp DESC LIMIT {limit}"
    results = conn.execute(base).fetchall()
    conn.close()

    payload = [
        {
            "id": r[0],
            "claim": r[1],
            "tags": r[2],
            "sub": r[3],
            "timestamp": r[4].isoformat(),  # ? fixed datetime serialization
            "content": r[5]
        }
        for r in results
    ]

    with open(RAM_CACHE_PATH, "w", encoding="utf-8") as f:
        json.dump(payload, f, indent=2)

    print(f"? RAM cache loaded with {len(payload)} fragments.")

if __name__ == "__main__":
    load_top_fragments()

==== reflection_agent.py ====
import os
import json
import uuid
from datetime import datetime
from tools.run_model import run_model

MODEL = "mistral:7b"
ROLE = "reflection"

LOG_FILE = "logs/generated_reports.jsonl"


def load_recent(n=10):
    with open(LOG_FILE, "r", encoding="utf-8") as f:
        lines = list(f.readlines())[-n:]
    entries = [json.loads(line) for line in lines]
    return entries
```

```
def reflect_on(entries):
    text = "\n\n".join(f"[{e['timestamp']}] {e['task']} ? {e['output'][:100]}..." for e in entries)
    prompt = f"""
Based on the following recent tasks and outputs, summarize:
- What the system is focused on
- If it's repeating itself
- What future tasks might help extend the system

{text}
"""
    result = run_model(MODEL, prompt)
    log_summary(prompt, result)
    return result


def log_summary(prompt, summary):
    with open("logs/reflection_log.jsonl", "a", encoding="utf-8") as f:
        f.write(json.dumps({
            "id": str(uuid.uuid4())[:8],
            "timestamp": datetime.utcnow().isoformat(),
            "prompt": prompt,
            "summary": summary
        }) + "\n")

if __name__ == "__main__":
    recent = load_recent()
    print(reflect_on(recent))

==== report_synthesizer.py ====
import json
from datetime import datetime
from pathlib import Path

SOURCE = "logs/report_writer_output.jsonl"
OUTPUT = "logs/final_report_bundle.json"


def synthesize():
    if not Path(SOURCE).exists():
        print("No report sections found.")
        return

    with open(SOURCE, "r", encoding="utf-8") as f:
        lines = f.readlines()

    entries = sorted([json.loads(l) for l in lines], key=lambda x: x["id"])

    full_report = "\n\n".join([f"### {e['task']}\n{e['output']}" for e in entries])
    final = {
        "timestamp": datetime.utcnow().isoformat(),
        "sections": len(entries),
        "content": full_report
    }
```

```
    with open(OUTPUT, "w", encoding="utf-8") as f:
        json.dump(final, f, indent=2)

    print(f"? Synthesized final report with {len(entries)} sections ? {OUTPUT}")


if __name__ == "__main__":
    synthesize()

==== report_writer_agent.py ====
import json
import uuid
from datetime import datetime
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.parallel_rag_query import query_all
from tools.run_model import run_model

MODEL = "mistral:7b"
ROLE = "report_writer"
QUEUE = "tasks/report_queue.json"
LOG = "logs/report_writer_output.jsonl"


def consume_queue():
    with open(QUEUE, "r", encoding="utf-8") as f:
        queue = json.load(f)

    for sid, task in queue.items():
        context = query_all(task)
        prompt = build_prompt_from_ram(task, context)
        output = run_model(MODEL, prompt)
        log_output(sid, task, output)

    print(f"? Wrote {len(queue)} report sections.")


def log_output(section_id, task, output):
    entry = {
        "id": section_id,
        "role": ROLE,
        "timestamp": datetime.utcnow().isoformat(),
        "task": task,
        "output": output
    }
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(entry) + "\n")

if __name__ == "__main__":
    consume_queue()

==== reranker_agent.py ====
import json
import uuid
```

```
from datetime import datetime
from tools.parallel_rag_query import query_all
from tools.llm_ram_prompt_builder import build_prompt_from_ram
from tools.run_model import run_model

MODEL = "phi:2"
ROLE = "reranker"
LOG = "logs/rerank_log.jsonl"


def rerank_fragments(query):
    context = query_all(query)
    scored = []
    for frag in context:
        score_prompt = f"On a scale of 0 to 1, how relevant is this to '{query}'?\n\n{frag['content']}"
        score = run_model(MODEL, score_prompt)
        try:
            score_val = float(score.strip())
        except:
            score_val = 0.0
        frag["relevance"] = round(score_val, 3)
        scored.append(frag)
    return sorted(scored, key=lambda x: x["relevance"], reverse=True)


def log_ranking(query, ranked):
    log_entry = {
        "id": str(uuid.uuid4())[:8],
        "timestamp": datetime.utcnow().isoformat(),
        "query": query,
        "top": [{"id": f["id"], "relevance": f["relevance"]} for f in ranked[:5]]
    }
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(log_entry) + "\n")


def run(query):
    ranked = rerank_fragments(query)
    log_ranking(query, ranked)
    print(f"\n[Top 3 for: '{query}']")
    for f in ranked[:3]:
        print(f"- ({f['relevance']}) {f['claim'][:80]}")


if __name__ == "__main__":
    import sys
    q = " ".join(sys.argv[1:]) or "what modules analyze system behavior"
    run(q)

==== server.py ====
# server.py ? Natasha, Online

from flask import Flask, request, render_template, jsonify
import subprocess
```

```
import os

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("search.html")

@app.route("/search")
def search():
    keyword = request.args.get("q", "")
    limit = request.args.get("limit", "10")

    if not keyword:
        return jsonify({"error": "Missing search keyword."}), 400

    try:
        result = subprocess.run(
            ["python", "C:/real_memory_system/tools/goal_router.py", keyword, limit],
            capture_output=True,
            text=True,
            check=True
        )

        lines = result.stdout.splitlines()
        results = []
        current = {}
        for line in lines:
            if line.startswith("[") and "]" in line:
                if current:
                    results.append(current)
                    current = {}
                current['category'] = line.split("]")[0][1:]
                current['filename'] = line.split("]")[1].strip()
            elif line.startswith("?"):
                current['filepath'] = line.replace("?", "").strip()
            else:
                current['snippet'] = current.get('snippet', '') + line.strip() + " "
        if current:
            results.append(current)

        return render_template("search.html", results=results, keyword=keyword)

    except subprocess.CalledProcessError as e:
        return render_template("search.html", error=e.stderr)

if __name__ == "__main__":
    app.run(debug=True)

==== start_brain_full.py ====
# start_brain_full.py (Clean Foundation - Verified Agent Stack with LLM Hook)

import subprocess
import threading
```

```
import time
import os
from pathlib import Path

# --- CLEAN CONFIG ---
BASE_DIR = Path("C:/real_memory_system")
TOOLS_DIR = BASE_DIR / "tools"

# --- VERIFIED AGENT SCRIPTS ---
LAUNCH_SCRIPTS = [
    ("OCR Media Indexer", TOOLS_DIR / "media_fragment_indexer.py"),
    ("Goal Router", TOOLS_DIR / "goal_router.py"),
    ("Intent Router", TOOLS_DIR / "intent_router_agent.py"),
    ("LLM Prompt Builder", TOOLS_DIR / "llm_ram_prompt_builder.py"),
    ("Agent Log Viewer", TOOLS_DIR / "agent_log_viewer.py"),
    ("AgentIQ Profiler", TOOLS_DIR / "agentiq_profiler_agent.py"),
    ("Auto Writer", TOOLS_DIR / "auto_writer.py"),
    ("Combine to PDF", TOOLS_DIR / "combine_to_pdf.py"),
    ("Compile to PDF", TOOLS_DIR / "compile_to_pdf.py"),
    ("Fragment Feedback Loop", TOOLS_DIR / "fragment_feedback_loop.py"),
    ("Model Broker", TOOLS_DIR / "multi_model_broker.py"),
    ("PDF SQL Retriever", TOOLS_DIR / "pdf_sql_retriever_agent.py"),
    ("Prompt Assembler", TOOLS_DIR / "prompt_assembler.py"),
    ("Query Context", TOOLS_DIR / "query_context.py"),
    ("Media to Memory", TOOLS_DIR / "media_to_memory.py"),
    ("Memory Log Viewer", TOOLS_DIR / "memory_log_viewer.py"),
    ("Memory Stack Boot", TOOLS_DIR / "memory_stack_boot.py"),
    ("Micro Agent Template", TOOLS_DIR / "micro_agent_template.py"),
    ("Multi-Brain Dispatcher", TOOLS_DIR / "multi_brain_dispatcher.py"),
    ("Report Synthesizer", TOOLS_DIR / "report_synthesizer.py"),
    ("Report Writer Agent", TOOLS_DIR / "report_writer_agent.py"),
    ("Reranker Agent", TOOLS_DIR / "reranker_agent.py"),
    ("Server", TOOLS_DIR / "server.py"),
    ("Structured Report Planner", TOOLS_DIR / "structured_report_planner.py"),
    ("RAG Agent Scheduler", TOOLS_DIR / "rag_agent_scheduler.py"),
    ("RAG to Ollama", TOOLS_DIR / "rag_to_ollama.py"),
    ("RAM Loader", TOOLS_DIR / "ram_loader.py"),
    ("Reflection Agent", TOOLS_DIR / "reflection_agent.py"),
    ("Super Sorter", TOOLS_DIR / "super_sorter_parallel.py"),
    ("Swarm Console", TOOLS_DIR / "swarm_console.py"),
    ("Swarm Control Panel", TOOLS_DIR / "swarm_control_panel.py"),
    ("Swarm VM Launcher", TOOLS_DIR / "swarm_vm_launcher.py"),
    ("Task Feedback Ranker", TOOLS_DIR / "task_feedback_ranker.py"),
    ("LLM Broker", TOOLS_DIR / "multi_model_broker.py")
]

def launch(label, path):
    if not path.exists():
        print(f"[SKIP] {label} missing: {path}")
        return
    def run():
        try:
            print(f"[BOOT] {label}...")
            subprocess.run(["python", str(path)], check=True)
```

```
        except Exception as e:
            print(f"[ERROR] {label} failed: {e}")
    thread = threading.Thread(target=run)
    thread.start()
    time.sleep(0.2)

def boot_stack():
    for label, script_path in LAUNCH_SCRIPTS:
        launch(label, script_path)
    print("\n[BOOT] Minimal swarm system launched.")

if __name__ == "__main__":
    try:
        boot_stack()
    except Exception as e:
        print(f"\n[ERROR] Boot failed: {e}")
    input("\n[Press Enter to close]")

==== start_swarm_upgraded.py ====
import subprocess
import threading
import time
import os

TOOLS = "tools"

components = [
    ("? RAM Loader", ["python", f"{TOOLS}/ram_loader.py"]),
    ("? RAG Scheduler", ["python", f"{TOOLS}/rag_agent_scheduler.py"]),
    ("? Agent Launcher", ["python", f"{TOOLS}/swarm_vm_launcher.py"]),
    ("? Reflection Agent", ["python", f"{TOOLS}/reflection_agent.py"]),
    ("? Intent Router", ["python", f"{TOOLS}/intent_router_agent.py"]),
    ("? OCR Media Indexer", ["python", f"{TOOLS}/media_fragment_indexer.py"]),
    ("? Report Writer", ["python", f"{TOOLS}/report_writer_agent.py"]),
    ("? Profiler", ["python", f"{TOOLS}/agentiq_profiler_agent.py"])
]

def launch(label, cmd):
    def runner():
        print(f"\n[BOOT] {label}...")
        subprocess.run(cmd)
    thread = threading.Thread(target=runner)
    thread.start()
    time.sleep(0.5)

if __name__ == "__main__":
    print("? [Swarm Boot Sequence: Upgraded Stack]")
    for label, cmd in components:
        launch(label, cmd)
    print("? All core swarm systems initialized.")

==== structured_report_planner.py ====
import uuid
import json
```

```
from datetime import datetime
from tools.run_model import run_model

MODEL = "mistral:7b"
ROLE = "report_planner"


def plan_sections(topic, structure):
    plan_prompt = f"""
You are a strategic AI agent. Your task is to plan a structured report.
Topic: {topic}
Structure: {structure}

Break this into a list of clear sections with titles and goals.
"""
    output = run_model(MODEL, plan_prompt)
    return output


def dispatch_sections(plan, topic):
    sections = [line for line in plan.split("\n") if line.strip() and "." in line]
    queued = {}
    for i, section in enumerate(sections):
        prompt = f"Write the section titled '{section}' for a report on {topic}."
        queued[f"section_{i+1}"] = prompt

    with open("tasks/report_queue.json", "w", encoding="utf-8") as f:
        json.dump(queued, f, indent=2)

    print(f"? Dispatched {len(queued)} sections to report queue.")


if __name__ == "__main__":
    import sys
    topic = sys.argv[1] if len(sys.argv) > 1 else "Symbolic Swarm Architecture"
     structure = sys.argv[2] if len(sys.argv) > 2 else "Introduction, Design Layers, Agent Stack, Retrieval,
Logic, Conclusion"
    plan = plan_sections(topic, structure)
    dispatch_sections(plan, topic)

==== super_sorter_parallel.py ====
import os
import hashlib
from pathlib import Path
from datetime import datetime
import shutil
import duckdb
from concurrent.futures import ThreadPoolExecutor, as_completed

FEED_DIR = Path("C:/real_memory_system/FEEDING_TIME")
FRAG_DIR = Path("C:/real_memory_system/fragments")
PRESERVE_DIR = Path("C:/real_memory_system/preserved")
LOG_PATH = Path("C:/real_memory_system/logs/sorting.log")
HASH_TRACKER = FRAG_DIR / "hashes"
```

```
CHUNK_SIZE = 5120
DB_PATH = Path("C:/real_memory_system/memory.duckdb")

CATEGORY_MAP = {
    ".py": ("code", "python"), ".yaml": ("config", "yaml"), ".yml": ("config", "yaml"),
    ".log": ("logs", "raw"), ".txt": ("notes", "text"), ".md": ("notes", "markdown"),
    ".json": ("data", "json"), ".html": ("web", "html"), ".csv": ("data", "csv"),
    ".duckdb": ("data", "duckdb"), ".db": ("data", "sqlite"),
    ".jpg": ("media", "images"), ".jpeg": ("media", "images"), ".png": ("media", "images"),
    ".webp": ("media", "images"), ".gif": ("media", "images"),
    ".mp4": ("media", "video"), ".mov": ("media", "video"),
    ".avi": ("media", "video"), ".mkv": ("media", "video"),
    ".exe": ("bin", "executables"), ".bin": ("bin", "binaries"),
    ".zip": ("bin", "archives"), ".tar": ("bin", "archives"), ".7z": ("bin", "archives")
}

def log(msg):
    LOG_PATH.parent.mkdir(parents=True, exist_ok=True)
    with open(LOG_PATH, "a", encoding="utf-8") as f:
        f.write(f"[{datetime.now().isoformat()}] {msg}\n")
    print(f"[SORTER] {msg}")

def hash_file(path):
    h = hashlib.sha1()
    with open(path, "rb") as f:
        while chunk := f.read(8192):
            h.update(chunk)
    return h.hexdigest()

def get_category_and_sub(path):
    return CATEGORY_MAP.get(path.suffix.lower(), ("misc", "unknown"))

def insert_into_db(fragment_id, claim, sub_category, tags, timestamp, filepath, content):
    conn = duckdb.connect(DB_PATH)
    conn.execute("""
        CREATE TABLE IF NOT EXISTS fragments (
            id TEXT PRIMARY KEY,
            claim TEXT,
            sub_category TEXT,
            confidence DOUBLE,
            tags TEXT[],
            timestamp TIMESTAMP,
            filepath TEXT,
            content TEXT
        )
    """)
    conn.execute("""
        INSERT INTO fragments (
            id, claim, sub_category, confidence, tags, timestamp, filepath, content
        ) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """, (fragment_id, claim, sub_category, 1.0, tags, timestamp, filepath, content))
    conn.close()

def write_chunk_file(chunk, cat, sub, base, i, source_path):
```

```
    chunk_hash = hashlib.sha1(chunk.encode("utf-8")).hexdigest()
    fname = f"{base}_part{i+1}_{chunk_hash[:8]}.txt"
    out_dir = FRAG_DIR / cat / sub
    out_dir.mkdir(parents=True, exist_ok=True)
    out_path = out_dir / fname

    if out_path.exists():
        log(f"Duplicate chunk skipped: {fname}")
        return

    with open(out_path, "w", encoding="utf-8") as out:
        out.write(chunk)

    tags = [cat, sub]
    ts = datetime.now().isoformat()
    insert_into_db(chunk_hash[:12], chunk[:80].strip(), sub, tags, ts, str(out_path), chunk)
    log(f"[OK] Wrote + Indexed: {out_path.name} ? {cat}/{sub}")

def preserve_file(path, cat, sub):
    target = PRESERVE_DIR / cat / sub / path.name
    target.parent.mkdir(parents=True, exist_ok=True)
    shutil.copy2(path, target)

    ts = datetime.now().isoformat()
    insert_into_db(
        hash_file(path)[:12],
        f"{path.name}",
        sub,
        [cat, sub],
        ts,
        str(target),
        None
    )
    log(f"[PRESERVED] {path.name} ? {cat}/{sub}")

def split_and_store(path):
    try:
        cat, sub = get_category_and_sub(path)
        if cat in ["media", "bin", "data"] and path.suffix.lower() not in [".json", ".csv", ".yaml", ".yml"]:
            preserve_file(path, cat, sub)
        else:
            with open(path, "r", encoding="utf-8", errors="ignore") as f:
                content = f.read()
            if len(content.strip()) < 5:
                log(f"[SKIP] Empty or junk file: {path.name}")
                return

            base = path.stem
            chunks = [content[i:i + CHUNK_SIZE] for i in range(0, len(content), CHUNK_SIZE)]
            for i, chunk in enumerate(chunks):
                write_chunk_file(chunk, cat, sub, base, i, path)

        path.unlink()
        log(f"[?] Removed processed file: {path.name}")
```

```
    except Exception as e:
        log(f"[ERROR] Failed to process {path.name}: {e}")

def run_super_sorter():
    HASH_TRACKER.mkdir(parents=True, exist_ok=True)
    all_files = []

    for f in FEED_DIR.rglob("*.*"):
        if f.is_file():
            file_hash = hash_file(f)
            hash_record = HASH_TRACKER / f"{file_hash}.ok"
            if not hash_record.exists():
                all_files.append((f, hash_record))

    with ThreadPoolExecutor(max_workers=16) as executor:
        futures = {executor.submit(split_and_store, f[0]): f[1] for f in all_files}
        for future in as_completed(futures):
            try:
                futures[future].touch()
            except Exception as e:
                log(f"[ERROR] Failed to touch hash record: {e}")

if __name__ == "__main__":
    run_super_sorter()

==== swarm_console.py ====
# swarm_console.py
import subprocess
import os
import time
from pathlib import Path

TOOLS_DIR = Path("C:/real_memory_system/tools")
LOG_DIR = Path("C:/real_memory_system/logs")
COMMANDS = {
    "sorter": "super_sorter_parallel.py",
    "ram": "ram_loader.py",
    "swarm": "multi_brain_dispatcher.py",
    "oi": [
        "interpreter",
        "--model", "ollama/codellama:34b-instruct",
        "--context_window", "65536",
        "--max_tokens", "4096",
        "-y"
    ]
}

def run(name):
    if name == "oi":
        print(f"? Launching Open Interpreter...")
        subprocess.Popen(COMMANDS["oi"])
    else:
        target = TOOLS_DIR / COMMANDS[name]
        if not target.exists():
```

```
            print(f"? Script not found: {target}")
            return
        print(f"? Running {name}...")
        subprocess.run(["python", str(target)], check=True)

def start_all():
    print("? Starting full swarm stack...\n")
    run("sorter")
    run("ram")
    run("oi")
    run("swarm")
    print("\n? Full system started.")

def check_status():
    print("? System Status\n")
    for log in LOG_DIR.glob("*.txt"):
        size_kb = round(log.stat().st_size / 1024, 2)
        print(f"? {log.name} ? {size_kb} KB ? Last modified: {time.ctime(log.stat().st_mtime)}")

def menu():
    while True:
        print("\n? Swarm Console")
        print("??????????????????????????????????")
        print("1. start all")
        print("2. start sorter")
        print("3. start ram")
        print("4. start swarm")
        print("5. start oi")
        print("6. status")
        print("7. exit")
        cmd = input("\n> ").strip().lower()

        if cmd in ["1", "start all"]:
            start_all()
        elif cmd in ["2", "start sorter"]:
            run("sorter")
        elif cmd in ["3", "start ram"]:
            run("ram")
        elif cmd in ["4", "start swarm"]:
            run("swarm")
        elif cmd in ["5", "start oi"]:
            run("oi")
        elif cmd in ["6", "status"]:
            check_status()
        elif cmd in ["7", "exit", "quit"]:
            print("? Exiting Swarm Console.")
            break
        else:
            print("? Unknown command.")

if __name__ == "__main__":
    menu()

==== swarm_control_panel.py ====
```

```
from nicegui import ui
import json
from pathlib import Path
import subprocess
import os

AGENTS = json.loads(Path("tiny_model_registry.json").read_text())
agent_status = {agent['id']: False for agent in AGENTS}
process_handles = {}

ui.label("? Swarm Control Panel").classes("text-2xl mb-4")

def toggle_agent(agent):
    aid = agent['id']
    if not agent_status[aid]:
        env = os.environ.copy()
        env['AGENT_ID'] = aid
        env['AGENT_ROLE'] = agent['role']
        env['AGENT_MODEL'] = agent['model']
        p = subprocess.Popen(["python", "micro_agent_template.py"], env=env)
        process_handles[aid] = p
        agent_status[aid] = True
    else:
        process_handles[aid].terminate()
        agent_status[aid] = False
    ui.run_javascript("location.reload()")

with ui.column():
    for agent in AGENTS:
        with ui.row():
            ui.label(f"{agent['role']} ? {agent['model']}")
            ui.button(
                "? Running" if agent_status[agent['id']] else "? Stopped",
                on_click=lambda a=agent: toggle_agent(a),
                color="green" if agent_status[agent['id']] else "grey",
            )

ui.run(title="Swarm Panel", native=False)

==== swarm_vm_launcher.py ====
# router_controller.py ? CLI + AgentIQ LLM Router

import asyncio
from parallel_rag_query import query_all
from llm_ram_prompt_builder import build_prompt_from_ram
from multi_model_broker import run_model

ROUTERS = {
    "summarize": "tinyllama:1.1b-q2_K",
    "code": "tinyllama:1.1b-q2_K",
    "search": "zephyr:7b-q3_K",
    "reason": "mistral:7b-q3_K",
    "report": "tinydolphin:1.1b-q2_K",
    "fallback": "phi:2"
```

```
}

def route(prompt):
    prompt = prompt.lower()
    if "summarize" in prompt:
        return ROUTERS["summarize"]
    if "fix" in prompt or "error" in prompt:
        return ROUTERS["code"]
    if "find" in prompt or "google" in prompt:
        return ROUTERS["search"]
    if "why" in prompt or "explain" in prompt:
        return ROUTERS["reason"]
    if "report" in prompt or "section" in prompt:
        return ROUTERS["report"]
    return ROUTERS["fallback"]

async def process_user_input(user_input):
    model = route(user_input)
    print(f"[? Router] ? Selected model: {model}")

    from parallel_rag_query import query_sources

    # Run all memory source queries in parallel
    memory_fragments = []
    for source in ["sql", "unstructured", "agent", "log", "ocr"]:
        memory_fragments += query_sources(user_input, source)
    prompt = build_prompt_from_ram(user_input, memory_fragments)

    try:
        result = await run_model(model, prompt)
    except Exception as e:
        result = f"[ERROR] model call failed: {e}"
    return result

if __name__ == "__main__":
    import sys
    user_query = " ".join(sys.argv[1:]) or "summarize this data stream"
    output = asyncio.run(process_user_input(user_query))
    print("\n[? Output]\n", output)

==== task_feedback_ranker.py ====
import json
from collections import defaultdict
from pathlib import Path

LOG_PATH = "logs/report_writer_output.jsonl"
RANK_PATH = "logs/task_feedback_scores.json"



def score_tasks():
    if not Path(LOG_PATH).exists():
        print("?? No logs found.")
        return
```

```
    task_counts = defaultdict(int)
    with open(LOG_PATH, "r", encoding="utf-8") as f:
        for line in f:
            try:
                task = json.loads(line)["task"]
                task_counts[task] += 1
            except:
                continue

    ranked = sorted(task_counts.items(), key=lambda x: x[1], reverse=True)
    with open(RANK_PATH, "w", encoding="utf-8") as f:
        json.dump(ranked, f, indent=2)

    print(f"? Ranked {len(ranked)} tasks by recurrence ? {RANK_PATH}")


if __name__ == "__main__":
    score_tasks()

==== tiny_model_registry.json ====
[
  {
    "id": "agent01",
    "role": "summarizer",
    "model": "tinydolphin:1.1b"
  },
  {
    "id": "agent02",
    "role": "coder",
    "model": "tinyllama:1.1b"
  },
  {
    "id": "agent03",
    "role": "retriever",
    "model": "phi:2"
  },
  {
    "id": "agent04",
    "role": "reasoner",
    "model": "zephyr:7b"
  },
  {
    "id": "agent05",
    "role": "generalist",
    "model": "mistral:7b"
  }
]

==== validate_fragment_links.py ====
import duckdb
from pathlib import Path

DB_DIR = Path("C:/real_memory_system/memory_db")
FRAGMENTS_ROOT = Path("C:/real_memory_system/fragments")
```

```
MISSING = []

for db_file in DB_DIR.glob("*.duckdb"):
    print(f"? Scanning: {db_file.name}")
    con = duckdb.connect(db_file)
    try:
        rows = con.execute("SELECT source_file FROM fragments").fetchall()
        for (path,) in rows:
            full_path = FRAGMENTS_ROOT / path
            if not full_path.exists():
                MISSING.append((db_file.name, str(full_path)))
    except Exception as e:
        print(f"?? Error reading {db_file}: {e}")
    finally:
        con.close()

if MISSING:
    print("\n? Broken References Found:")
    for db, path in MISSING:
        print(f"[{db}] ? {path}")
else:
    print("\n? All DB entries point to real files.")
```

```
==== __init__.py ====
from .test_metadata import *

==== __init__1.py ====

==== __init__2.py ====

==== __init__3.py ====

==== adaptive_installer.py ====
# adaptive_installer.py
import os
import yaml
import psutil
import platform
from pathlib import Path
from shutil import disk_usage

BASE = Path(__file__).parent
CONFIG_PATH = BASE / "system_config.yaml"

def detect_disks():
    disks = []
    for part in psutil.disk_partitions():
        try:
            usage = disk_usage(part.mountpoint)
            disks.append({
                'mount': part.mountpoint,
                'fstype': part.fstype,
                'free_gb': round(usage.free / 1e9, 2),
                'total_gb': round(usage.total / 1e9, 2)
            })
        except Exception:
            continue
    return disks

def choose_primary_mount(disks):
    if not disks:
        return "C:\\" if platform.system() == "Windows" else "/"
    return disks[0]['mount']

def detect_tier(profile):
    ram = profile['ram_total_mb']
    cores = profile['threads']
    has_nvme = any(d['mount'][0].upper() in ['C', 'D', 'F'] for d in profile['disks'])

    if ram < 8000 or cores < 2:
        return "tier_0_minimal"
    elif ram < 16000:
        return "tier_1_agent"
    elif ram < 64000:
        return "tier_2_blade"
    elif ram >= 64000 and has_nvme:
```

```
        return "tier_3_controller"
    else:
        return "tier_unknown"

def generate_config(profile):
    return {
        'platform': profile['os'],
        'cpu_cores': profile['cpu_cores'],
        'threads': profile['threads'],
        'ram_total': profile['ram_total_mb'],
        'ram_available': profile['ram_available_mb'],
        'disk_primary_mount': choose_primary_mount(profile['disks']),
        'disk_free_total': profile['disk_free'],
        'logic_tier': detect_tier(profile),
        'logic_root': 'C:/logicshred/' if profile['os'] == "Windows" else '/neurostore/',
        'logic_ram': {},
        'fragment_defaults': {
            'emotion_decay': True,
            'mutation_rate': 'auto'
        }
    }

def get_profile():
    ram_total = psutil.virtual_memory().total
    ram_available = psutil.virtual_memory().available
    cpu_count = psutil.cpu_count(logical=False)
    cpu_threads = psutil.cpu_count()
    disks = detect_disks()
    total_disk_free = sum([d['free_gb'] for d in disks])

    return {
        'os': platform.system(),
        'cpu_cores': cpu_count,
        'threads': cpu_threads,
        'ram_total_mb': round(ram_total / 1e6),
        'ram_available_mb': round(ram_available / 1e6),
        'disks': disks,
        'disk_free': round(total_disk_free, 2)
    }

def write_yaml(config):
    with open(CONFIG_PATH, 'w') as f:
        yaml.safe_dump(config, f)

def main():
    print("[auto_configurator] INFO Detected system profile:")
    profile = get_profile()
    for k, v in profile.items():
        print(f"  - {k}: {v}")
    config = generate_config(profile)
    write_yaml(config)
    print(f"[auto_configurator] [OK] Config written to {CONFIG_PATH.name}")
    print(f"[auto_configurator] INFO System optimized by logic profile: {config['logic_tier']}")
```

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

==== async_swarm_launcher.py ====
import asyncio
import subprocess

TASKS = [
    "fragment_decay_engine.py",
    "dreamwalker.py",
    "validator.py",
    "mutation_engine.py"
]

async def run_script(name):
    proc = await asyncio.create_subprocess_exec("python", name)
    await proc.wait()

async def main():
    coros = [run_script(task) for task in TASKS]
    await asyncio.gather(*coros)

if __name__ == "__main__":
    asyncio.run(main())

==== auto_configurator.backup.py ====
"""
LOGICSHREDDER :: auto_configurator.py
Purpose: Scan system and assign optimal config values based on hardware
"""

import psutil
import yaml
from pathlib import Path
import platform
import time

CONFIG_PATH = Path("configs/system_config.yaml")
BACKUP_PATH = CONFIG_PATH.with_suffix(".autobackup.yaml")

def get_system_profile():
    return {
        "cores": psutil.cpu_count(logical=False),
        "threads": psutil.cpu_count(logical=True),
        "total_ram": round(psutil.virtual_memory().total / (1024**2)),  # in MB
        "available_ram": round(psutil.virtual_memory().available / (1024**2)),
        "disk_free": round(psutil.disk_usage(".").free / (1024**3)),  # in GB
        "platform": platform.system()
    }

def generate_config(profile):
    cfg = {
        "brain": {
            "name": "LOGICSHREDDER",
```

```
    "version": 1.0,
    "allow_mutation": profile["cores"] >= 2,
    "allow_cold_storage": True,
    "auto_snapshot": profile["disk_free"] >= 5,
    "emotion_enabled": profile["total_ram"] >= 4000,
    "safe_mode": False,
    "lock_respect": True,
    "optimized_by": "auto_configurator"
},
"resources": {
    "cpu_limit_percent": 90 if profile["cores"] >= 4 else 70,
    "min_ram_mb": 2048 if profile["total_ram"] < 4000 else 4096,
    "io_watchdog_enabled": True,
    "max_io_mb_per_minute": 300 if profile["disk_free"] < 2 else 500
},
"paths": {
    "fragments": "fragments/core/",
    "archive": "fragments/archive/",
    "overflow": "fragments/overflow/",
    "cold": "fragments/cold/",
    "logs": "logs/",
    "profiler": "logs/agent_stats/",
    "snapshot": "snapshots/",
    "input": "input/"
},
"agents": {
    "token_agent": True,
    "guffifier": True,
    "mutation_engine": profile["cores"] >= 2,
    "validator": True,
    "dreamwalker": profile["cores"] >= 4,
    "cold_logic_mover": True,
    "meta_agent": True,
    "cortex_logger": True,
    "heatmap": profile["total_ram"] >= 3000
},
"security": {
    "auto_lock_on_snapshot": True,
    "forbid_external_io": True,
    "network_disabled": True,
    "write_protect_brain": False
},
"modes": {
    "verbose_logging": True,
    "batch_mode": False,
    "ignore_errors": False,
    "dry_run": False
},
"tuning": {
    "contradiction_sensitivity": 0.8,
    "decay_rate": 0.02 if profile["cores"] >= 4 else 0.04,
    "mutation_aggression": 0.7 if profile["cores"] >= 4 else 0.4,
    "rewalk_threshold": 0.6,
    "cold_logic_threshold": 0.3,
```

```
            "curiosity_bias": 0.3 if profile["available_ram"] > 3000 else 0.1
        }
    }
    return cfg

def write_config(cfg):
    if CONFIG_PATH.exists():
        CONFIG_PATH.replace(BACKUP_PATH)
    with open(CONFIG_PATH, 'w', encoding='utf-8') as out:
        yaml.dump(cfg, out)
    print(f"[auto_configurator] [OK] Config written to {CONFIG_PATH.name}")
    print(f"[auto_configurator] INFO System optimized by logic profile.")

def main():
    profile = get_system_profile()
    print("[auto_configurator] INFO Detected system profile:")
    for k, v in profile.items():
        print(f"  - {k}: {v}")
    config = generate_config(profile)
    write_config(config)

if __name__ == "__main__":
    main()

==== auto_configurator.backup2.py ====
"""
LOGICSHREDDER :: auto_configurator.py
Purpose: Scan system and assign optimal config values based on hardware
"""

import psutil
import yaml
from pathlib import Path
import platform
import time

CONFIG_PATH = Path("configs/system_config.yaml")
BACKUP_PATH = CONFIG_PATH.with_suffix(".autobackup.yaml")


def get_system_profile():
    disks = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            disks.append({
                "mount": part.mountpoint,
                "fstype": part.fstype,
                "free_gb": round(usage.free / (1024**3), 2),
                "total_gb": round(usage.total / (1024**3), 2)
            })
        except PermissionError:
            continue
```

```
    total_free = sum(d['free_gb'] for d in disks)
    primary = disks[0] if disks else {"mount": "?", "free_gb": 0}

    return {
        "cores": psutil.cpu_count(logical=False),
        "threads": psutil.cpu_count(logical=True),
        "total_ram": round(psutil.virtual_memory().total / (1024**2)),
        "available_ram": round(psutil.virtual_memory().available / (1024**2)),
        "disks": disks,
        "disk_free_total": round(total_free, 2),
        "disk_primary_mount": primary["mount"],
        "platform": platform.system()
    }

def generate_config(profile):
    cfg = {
        "brain": {
            "name": "LOGICSHREDDER",
            "version": 1.0,
            "allow_mutation": profile["cores"] >= 2,
            "allow_cold_storage": True,
            "auto_snapshot": profile["disk_free_total"] >= 5,
            "emotion_enabled": profile["total_ram"] >= 4000,
            "safe_mode": False,
            "lock_respect": True,
            "optimized_by": "auto_configurator"
        },
        "resources": {
            "cpu_limit_percent": 90 if profile["cores"] >= 4 else 70,
            "min_ram_mb": 2048 if profile["total_ram"] < 4000 else 4096,
            "io_watchdog_enabled": True,
            "max_io_mb_per_minute": 300 if profile["disk_free"] < 2 else 500
        },
        "paths": {
            "fragments": "fragments/core/",
            "archive": "fragments/archive/",
            "overflow": "fragments/overflow/",
            "cold": "fragments/cold/",
            "logs": "logs/",
            "profiler": "logs/agent_stats/",
            "snapshot": "snapshots/",
            "input": "input/"
        },
        "agents": {
            "token_agent": True,
            "guffifier": True,
            "mutation_engine": profile["cores"] >= 2,
            "validator": True,
            "dreamwalker": profile["cores"] >= 4,
            "cold_logic_mover": True,
            "meta_agent": True,
            "cortex_logger": True,
            "heatmap": profile["total_ram"] >= 3000
        },
```

```
        "security": {
            "auto_lock_on_snapshot": True,
            "forbid_external_io": True,
            "network_disabled": True,
            "write_protect_brain": False
        },
        "modes": {
            "verbose_logging": True,
            "batch_mode": False,
            "ignore_errors": False,
            "dry_run": False
        },
        "tuning": {
            "contradiction_sensitivity": 0.8,
            "decay_rate": 0.02 if profile["cores"] >= 4 else 0.04,
            "mutation_aggression": 0.7 if profile["cores"] >= 4 else 0.4,
            "rewalk_threshold": 0.6,
            "cold_logic_threshold": 0.3,
            "curiosity_bias": 0.3 if profile["available_ram"] > 3000 else 0.1
        }
    }
    return cfg

def write_config(cfg):
    if CONFIG_PATH.exists():
        CONFIG_PATH.replace(BACKUP_PATH)
    with open(CONFIG_PATH, 'w', encoding='utf-8') as out:
        yaml.dump(cfg, out)
    print(f"[auto_configurator] [OK] Config written to {CONFIG_PATH.name}")
    print(f"[auto_configurator] INFO System optimized by logic profile.")

def main():
    profile = get_system_profile()
    print("[auto_configurator] INFO Detected system profile:")
    for k, v in profile.items():
        print(f"  - {k}: {v}")
    config = generate_config(profile)
    write_config(config)

if __name__ == "__main__":
    main()

==== auto_configurator.py ====
"""
LOGICSHREDDER :: auto_configurator.py
Purpose: Scan system and assign optimal config values based on hardware
"""

import psutil
import yaml
from pathlib import Path
import platform
import time
```

```
CONFIG_PATH = Path("configs/system_config.yaml")
BACKUP_PATH = CONFIG_PATH.with_suffix(".autobackup.yaml")


def get_system_profile():
    disks = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            disks.append({
                "mount": part.mountpoint,
                "fstype": part.fstype,
                "free_gb": round(usage.free / (1024**3), 2),
                "total_gb": round(usage.total / (1024**3), 2)
            })
        except PermissionError:
            continue

    total_free = sum(d['free_gb'] for d in disks)
    primary = disks[0] if disks else {"mount": "?", "free_gb": 0}

    return {
        "cores": psutil.cpu_count(logical=False),
        "threads": psutil.cpu_count(logical=True),
        "total_ram": round(psutil.virtual_memory().total / (1024**2)),
        "available_ram": round(psutil.virtual_memory().available / (1024**2)),
        "disks": disks,
        "disk_free_total": round(total_free, 2),
        "disk_primary_mount": primary["mount"],
        "platform": platform.system()
    }

def generate_config(profile):
    cfg = {
        "brain": {
            "name": "LOGICSHREDDER",
            "version": 1.0,
            "allow_mutation": profile["cores"] >= 2,
            "allow_cold_storage": True,
            "auto_snapshot": profile["disk_free_total"] >= 5,
            "emotion_enabled": profile["total_ram"] >= 4000,
            "safe_mode": False,
            "lock_respect": True,
            "optimized_by": "auto_configurator"
        },
        "resources": {
            "cpu_limit_percent": 90 if profile["cores"] >= 4 else 70,
            "min_ram_mb": 2048 if profile["total_ram"] < 4000 else 4096,
            "io_watchdog_enabled": True,
            "max_io_mb_per_minute": 300 if profile["disk_free_total"] < 2 else 500
        },
        "paths": {
            "fragments": "fragments/core/",
            "archive": "fragments/archive/",
```

```
            "overflow": "fragments/overflow/",
            "cold": "fragments/cold/",
            "logs": "logs/",
            "profiler": "logs/agent_stats/",
            "snapshot": "snapshots/",
            "input": "input/"
        },
        "agents": {
            "token_agent": True,
            "guffifier": True,
            "mutation_engine": profile["cores"] >= 2,
            "validator": True,
            "dreamwalker": profile["cores"] >= 4,
            "cold_logic_mover": True,
            "meta_agent": True,
            "cortex_logger": True,
            "heatmap": profile["total_ram"] >= 3000
        },
        "security": {
            "auto_lock_on_snapshot": True,
            "forbid_external_io": True,
            "network_disabled": True,
            "write_protect_brain": False
        },
        "modes": {
            "verbose_logging": True,
            "batch_mode": False,
            "ignore_errors": False,
            "dry_run": False
        },
        "tuning": {
            "contradiction_sensitivity": 0.8,
            "decay_rate": 0.02 if profile["cores"] >= 4 else 0.04,
            "mutation_aggression": 0.7 if profile["cores"] >= 4 else 0.4,
            "rewalk_threshold": 0.6,
            "cold_logic_threshold": 0.3,
            "curiosity_bias": 0.3 if profile["available_ram"] > 3000 else 0.1
        }
    }
    return cfg

def write_config(cfg):
    if CONFIG_PATH.exists():
        CONFIG_PATH.replace(BACKUP_PATH)
    with open(CONFIG_PATH, 'w', encoding='utf-8') as out:
        yaml.dump(cfg, out)
    print(f"[auto_configurator] [OK] Config written to {CONFIG_PATH.name}")
    print(f"[auto_configurator] INFO System optimized by logic profile.")

def main():
    profile = get_system_profile()
    print("[auto_configurator] INFO Detected system profile:")
    for k, v in profile.items():
        print(f"  - {k}: {v}")
```

```
    config = generate_config(profile)
    write_config(config)

if __name__ == "__main__":
    main()

==== auto_configurator.repaired.py ====
"""
LOGICSHREDDER :: auto_configurator.py
Purpose: Scan system and assign optimal config values based on hardware
"""

import psutil
import yaml
from pathlib import Path
import platform
import time

CONFIG_PATH = Path("configs/system_config.yaml")
BACKUP_PATH = CONFIG_PATH.with_suffix(".autobackup.yaml")


def get_system_profile():
    disks = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            disks.append({
                "mount": part.mountpoint,
                "fstype": part.fstype,
                "free_gb": round(usage.free / (1024**3), 2),
                "total_gb": round(usage.total / (1024**3), 2)
            })
        except PermissionError:
            continue

    total_free = sum(d['free_gb'] for d in disks)
    primary = disks[0] if disks else {"mount": "?", "free_gb": 0}

    return {
        "cores": psutil.cpu_count(logical=False),
        "threads": psutil.cpu_count(logical=True),
        "total_ram": round(psutil.virtual_memory().total / (1024**2)),
        "available_ram": round(psutil.virtual_memory().available / (1024**2)),
        "disks": disks,
        "disk_free_total_total": round(total_free, 2),
        "disk_primary_mount": primary["mount"],
        "platform": platform.system()
    }

def generate_config(profile):
    cfg = {
        "brain": {
            "name": "LOGICSHREDDER",
```

```
    "version": 1.0,
    "allow_mutation": profile["cores"] >= 2,
    "allow_cold_storage": True,
    "auto_snapshot": profile["disk_free_total_total"] >= 5,
    "emotion_enabled": profile["total_ram"] >= 4000,
    "safe_mode": False,
    "lock_respect": True,
    "optimized_by": "auto_configurator"
},
"resources": {
    "cpu_limit_percent": 90 if profile["cores"] >= 4 else 70,
    "min_ram_mb": 2048 if profile["total_ram"] < 4000 else 4096,
    "io_watchdog_enabled": True,
    "max_io_mb_per_minute": 300 if profile["disk_free_total"] < 2 else 500
},
"paths": {
    "fragments": "fragments/core/",
    "archive": "fragments/archive/",
    "overflow": "fragments/overflow/",
    "cold": "fragments/cold/",
    "logs": "logs/",
    "profiler": "logs/agent_stats/",
    "snapshot": "snapshots/",
    "input": "input/"
},
"agents": {
    "token_agent": True,
    "guffifier": True,
    "mutation_engine": profile["cores"] >= 2,
    "validator": True,
    "dreamwalker": profile["cores"] >= 4,
    "cold_logic_mover": True,
    "meta_agent": True,
    "cortex_logger": True,
    "heatmap": profile["total_ram"] >= 3000
},
"security": {
    "auto_lock_on_snapshot": True,
    "forbid_external_io": True,
    "network_disabled": True,
    "write_protect_brain": False
},
"modes": {
    "verbose_logging": True,
    "batch_mode": False,
    "ignore_errors": False,
    "dry_run": False
},
"tuning": {
    "contradiction_sensitivity": 0.8,
    "decay_rate": 0.02 if profile["cores"] >= 4 else 0.04,
    "mutation_aggression": 0.7 if profile["cores"] >= 4 else 0.4,
    "rewalk_threshold": 0.6,
    "cold_logic_threshold": 0.3,
```

```
            "curiosity_bias": 0.3 if profile["available_ram"] > 3000 else 0.1
        }
    }
    return cfg

def write_config(cfg):
    if CONFIG_PATH.exists():
        CONFIG_PATH.replace(BACKUP_PATH)
    with open(CONFIG_PATH, 'w', encoding='utf-8') as out:
        yaml.dump(cfg, out)
    print(f"[auto_configurator] [OK] Config written to {CONFIG_PATH.name}")
    print(f"[auto_configurator] INFO System optimized by logic profile.")

def main():
    profile = get_system_profile()
    print("[auto_configurator] INFO Detected system profile:")
    for k, v in profile.items():
        print(f"  - {k}: {v}")
    config = generate_config(profile)
    write_config(config)

if __name__ == "__main__":
    main()

==== backup_and_export.py ====
import os
import tarfile
from datetime import datetime

EXPORT_DIR = os.path.expanduser("~/neurostore/backups")
SOURCE_DIRS = ["agents", "fragments", "logs", "meta", "runtime", "data"]

os.makedirs(EXPORT_DIR, exist_ok=True)
backup_name = f"neurostore_brain_{datetime.now().strftime('%Y%m%d_%H%M%S')}.tar.gz"
backup_path = os.path.join(EXPORT_DIR, backup_name)

with tarfile.open(backup_path, "w:gz") as tar:
    for folder in SOURCE_DIRS:
        if os.path.exists(folder):
            print(f"[+] Archiving {folder}/")
            tar.add(folder, arcname=folder)
        else:
            print(f"[-] Skipped missing folder: {folder}")

print(f"[OK] Brain backup complete -> {backup_path}")

==== belief_ingestor.py ====
"""
LOGICSHREDDER :: belief_ingestor.py
Purpose: Scans feedbox/ for .txt/.json/.yaml/.py files, extracts claims, converts to fragment YAMLs
"""

import os, uuid, yaml, json
from pathlib import Path
```

```
import time

FEED_DIR = Path("feedbox")
CONSUMED_DIR = FEED_DIR / "consumed"
FRAG_DIR = Path("fragments/core")

FEED_DIR.mkdir(exist_ok=True)
CONSUMED_DIR.mkdir(exist_ok=True)
FRAG_DIR.mkdir(parents=True, exist_ok=True)

def extract_claims_from_txt(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        lines = [line.strip("- ").strip() for line in f.readlines() if line.strip()]
    return [line for line in lines if len(line) > 5]

def extract_claims_from_json(file_path):
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)
        if isinstance(data, list):
            return [str(item).strip() for item in data if isinstance(item, str)]
        return []
    except:
        return []

def extract_claims_from_yaml(file_path):
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            data = yaml.safe_load(f)
        if isinstance(data, list):
            return [str(item).strip() for item in data if isinstance(item, str)]
        return []
    except:
        return []

def extract_claims_from_py(file_path):
    lines = []
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            if "==" in line or "def " in line or "return" in line:
                lines.append(line.strip())
    return lines

def write_fragment(claim):
    frag = {
        "id": str(uuid.uuid4())[:8],
        "claim": claim,
        "confidence": 0.85,
        "emotion": {},
        "timestamp": int(time.time())
    }
    fpath = FRAG_DIR / f"{frag['id']}.yaml"
    with open(fpath, 'w', encoding='utf-8') as f:
        yaml.safe_dump(frag, f)
```

```
def ingest_feedbox():
    files = list(FEED_DIR.glob("*.*"))
    total_claims = 0

    for file_path in files:
        claims = []
        ext = file_path.suffix.lower()

        if ext == ".txt":
            claims = extract_claims_from_txt(file_path)
        elif ext == ".json":
            claims = extract_claims_from_json(file_path)
        elif ext == ".yaml":
            claims = extract_claims_from_yaml(file_path)
        elif ext == ".py":
            claims = extract_claims_from_py(file_path)

        if claims:
            for claim in claims:
                write_fragment(claim)
            file_path.rename(CONSUMED_DIR / file_path.name)
            print(f"[ingestor] [OK] Ingested {len(claims)} from {file_path.name}")
            total_claims += len(claims)
        else:
            print(f"[ingestor] WARNING Skipped {file_path.name} (no claims)")

    print(f"[ingestor] INFO Total beliefs ingested: {total_claims}")

if __name__ == "__main__":
    ingest_feedbox()

==== bench.py ====
from __future__ import annotations

import argparse
import json
import os
import re
import signal
import socket
import subprocess
import sys
import threading
import time
import traceback
from contextlib import closing
from datetime import datetime

import matplotlib
import matplotlib.dates
import matplotlib.pyplot as plt
import requests
from statistics import mean
```

```
def main(args_in: list[str] | None = None) -> None:
    parser = argparse.ArgumentParser(description="Start server benchmark scenario")
    parser.add_argument("--name", type=str, help="Bench name", required=True)
    parser.add_argument("--runner-label", type=str, help="Runner label", required=True)
    parser.add_argument("--branch", type=str, help="Branch name", default="detached")
    parser.add_argument("--commit", type=str, help="Commit name", default="dirty")
    parser.add_argument("--host", type=str, help="Server listen host", default="0.0.0.0")
    parser.add_argument("--port", type=int, help="Server listen host", default="8080")
       parser.add_argument("--model-path-prefix", type=str, help="Prefix where to store the model files",
default="models")
    parser.add_argument("--n-prompts", type=int,
                            help="SERVER_BENCH_N_PROMPTS: total prompts to randomly select in the benchmark",
required=True)
    parser.add_argument("--max-prompt-tokens", type=int,
                             help="SERVER_BENCH_MAX_PROMPT_TOKENS: maximum prompt tokens to filter out in the
dataset",
                        required=True)
    parser.add_argument("--max-tokens", type=int,
                            help="SERVER_BENCH_MAX_CONTEXT: maximum context size of the completions request to
filter out in the dataset: prompt + predicted tokens",
                        required=True)
    parser.add_argument("--hf-repo", type=str, help="Hugging Face model repository", required=True)
    parser.add_argument("--hf-file", type=str, help="Hugging Face model file", required=True)
       parser.add_argument("-ngl", "--n-gpu-layers", type=int, help="layers to the GPU for computation",
required=True)
    parser.add_argument("--ctx-size", type=int, help="Set the size of the prompt context", required=True)
       parser.add_argument("--parallel", type=int, help="Set the number of slots for process requests",
required=True)
        parser.add_argument("--batch-size", type=int, help="Set the batch size for prompt processing",
required=True)
    parser.add_argument("--ubatch-size", type=int, help="physical maximum batch size", required=True)
    parser.add_argument("--scenario", type=str, help="Scenario to run", required=True)
    parser.add_argument("--duration", type=str, help="Bench scenario", required=True)

    args = parser.parse_args(args_in)

    start_time = time.time()

    # Start the server and performance scenario
    try:
        server_process = start_server(args)
    except Exception:
        print("bench: server start error :")
        traceback.print_exc(file=sys.stdout)
        sys.exit(1)

    # start the benchmark
    iterations = 0
    data = {}
    try:
        start_benchmark(args)
```

```
        with open("results.github.env", 'w') as github_env:
            # parse output
            with open('k6-results.json', 'r') as bench_results:
                # Load JSON data from file
                data = json.load(bench_results)
                for metric_name in data['metrics']:
                    for metric_metric in data['metrics'][metric_name]:
                        value = data['metrics'][metric_name][metric_metric]
                        if isinstance(value, float) or isinstance(value, int):
                            value = round(value, 2)
                            data['metrics'][metric_name][metric_metric]=value
                            github_env.write(

f"{escape_metric_name(metric_name)}_{escape_metric_name(metric_metric)}={value}\n")
                iterations = data['root_group']['checks']['success completion']['passes']

    except Exception:
        print("bench: error :")
        traceback.print_exc(file=sys.stdout)

    # Stop the server
    if server_process:
        try:
            print(f"bench: shutting down server pid={server_process.pid} ...")
            if os.name == 'nt':
                interrupt = signal.CTRL_C_EVENT
            else:
                interrupt = signal.SIGINT
            server_process.send_signal(interrupt)
            server_process.wait(0.5)

        except subprocess.TimeoutExpired:
            print(f"server still alive after 500ms, force-killing pid={server_process.pid} ...")
            server_process.kill()  # SIGKILL
            server_process.wait()

        while is_server_listening(args.host, args.port):
            time.sleep(0.1)

    title = (f"llama.cpp {args.name} on {args.runner_label}\n "
             f"duration={args.duration} {iterations} iterations")
    xlabel = (f"{args.hf_repo}/{args.hf_file}\n"
                                f"parallel={args.parallel}  ctx-size={args.ctx_size}  ngl={args.n_gpu_layers}
batch-size={args.batch_size}            ubatch-size={args.ubatch_size}            pp={args.max_prompt_tokens}
pp+tg={args.max_tokens}\n"
              f"branch={args.branch} commit={args.commit}")

    # Prometheus
    end_time = time.time()
    prometheus_metrics = {}
    if is_server_listening("0.0.0.0", 9090):
        metrics = ['prompt_tokens_seconds', 'predicted_tokens_seconds',
                   'kv_cache_usage_ratio', 'requests_processing', 'requests_deferred']
```

```
        for metric in metrics:
            resp = requests.get(f"http://localhost:9090/api/v1/query_range",
                                params={'query': 'llamacpp:' + metric, 'start': start_time, 'end': end_time,
'step': 2})

            with open(f"{metric}.json", 'w') as metric_json:
                metric_json.write(resp.text)

            if resp.status_code != 200:
                print(f"bench: unable to extract prometheus metric {metric}: {resp.text}")
            else:
                metric_data = resp.json()
                values = metric_data['data']['result'][0]['values']
                timestamps, metric_values = zip(*values)
                metric_values = [float(value) for value in metric_values]
                prometheus_metrics[metric] = metric_values
                timestamps_dt = [str(datetime.fromtimestamp(int(ts))) for ts in timestamps]
                plt.figure(figsize=(16, 10), dpi=80)
                plt.plot(timestamps_dt, metric_values, label=metric)
                plt.xticks(rotation=0, fontsize=14, horizontalalignment='center', alpha=.7)
                plt.yticks(fontsize=12, alpha=.7)

                ylabel = f"llamacpp:{metric}"
                plt.title(title,
                          fontsize=14, wrap=True)
                plt.grid(axis='both', alpha=.3)
                plt.ylabel(ylabel, fontsize=22)
                plt.xlabel(xlabel, fontsize=14, wrap=True)
                plt.gca().xaxis.set_major_locator(matplotlib.dates.MinuteLocator())
                plt.gca().xaxis.set_major_formatter(matplotlib.dates.DateFormatter("%Y-%m-%d %H:%M:%S"))
                plt.gcf().autofmt_xdate()

                # Remove borders
                plt.gca().spines["top"].set_alpha(0.0)
                plt.gca().spines["bottom"].set_alpha(0.3)
                plt.gca().spines["right"].set_alpha(0.0)
                plt.gca().spines["left"].set_alpha(0.3)

                # Save the plot as a jpg image
                plt.savefig(f'{metric}.jpg', dpi=60)
                plt.close()

                # Mermaid format in case images upload failed
                with open(f"{metric}.mermaid", 'w') as mermaid_f:
                    mermaid = (
                    f"""---
config:
    xyChart:
        titleFontSize: 12
        width: 900
        height: 600
    themeVariables:
        xyChart:
            titleColor: "#000000"
```

```
---
xychart-beta
    title "{title}"
    y-axis "llamacpp:{metric}"
    x-axis "llamacpp:{metric}" {int(min(timestamps))} --> {int(max(timestamps))}
    line [{', '.join([str(round(float(value), 2)) for value in metric_values])}]
                """)
                mermaid_f.write(mermaid)

    # 140 chars max for commit status description
    bench_results = {
        "i": iterations,
        "req": {
            "p95": round(data['metrics']["http_req_duration"]["p(95)"], 2),
            "avg": round(data['metrics']["http_req_duration"]["avg"], 2),
        },
        "pp": {
            "p95": round(data['metrics']["llamacpp_prompt_processing_second"]["p(95)"], 2),
            "avg": round(data['metrics']["llamacpp_prompt_processing_second"]["avg"], 2),
               "0": round(mean(prometheus_metrics['prompt_tokens_seconds']), 2) if 'prompt_tokens_seconds' in
prometheus_metrics else 0,
        },
        "tg": {
            "p95": round(data['metrics']["llamacpp_tokens_second"]["p(95)"], 2),
            "avg": round(data['metrics']["llamacpp_tokens_second"]["avg"], 2),
              "0": round(mean(prometheus_metrics['predicted_tokens_seconds']), 2) if 'predicted_tokens_seconds'
in prometheus_metrics else 0,
        },
    }
    with open("results.github.env", 'a') as github_env:
        github_env.write(f"BENCH_RESULTS={json.dumps(bench_results, indent=None, separators=(',', ':') )}\n")
        github_env.write(f"BENCH_ITERATIONS={iterations}\n")

        title = title.replace('\n', ' ')
        xlabel = xlabel.replace('\n', ' ')
        github_env.write(f"BENCH_GRAPH_TITLE={title}\n")
        github_env.write(f"BENCH_GRAPH_XLABEL={xlabel}\n")


def start_benchmark(args):
    k6_path = './k6'
    if 'BENCH_K6_BIN_PATH' in os.environ:
        k6_path = os.environ['BENCH_K6_BIN_PATH']
    k6_args = [
        'run', args.scenario,
        '--no-color',
        '--no-connection-reuse',
        '--no-vu-connection-reuse',
    ]
    k6_args.extend(['--duration', args.duration])
    k6_args.extend(['--iterations', args.n_prompts])
    k6_args.extend(['--vus', args.parallel])
    k6_args.extend(['--summary-export', 'k6-results.json'])
    k6_args.extend(['--out', 'csv=k6-results.csv'])
```