Project H.U.G.H.: A Neurosymbolic Agentic OS Implementation Blueprint

Phase 0: System Bedrock: Environment Setup and Tooling

This initial phase establishes the foundational hardware and software bedrock for the H.U.G.H. symbiote. All subsequent layers depend on the precise configuration of this environment.

0.1. Hardware Constraint Analysis: The "Local Suit"

The specified hardware, a Mid-2012 13-inch MacBook Pro (MacBookPro9,2) , dictates the primary technical constraint of this architecture. This model features a dual-core 22nm "Ivy Bridge" Intel processor (either a Core i5-3210M or i7-3230M).
The critical determination from this hardware profile is the graphics processor. The 13-inch model exclusively contains integrated Intel HD Graphics 4000. It does *not* possess a discrete NVIDIA GPU, which was limited to the 15-inch models of that era.
This analysis resolves the "CUDA-if-applicable" query: **CUDA is not applicable**.
This "No-CUDA Mandate" is the single most important architectural driver. All model inference for the Interface, Knowledge, and Agentic layers must be performed exclusively on the Ivy Bridge CPU. This mandates a software stack built on highly quantized models (GGUF) and C++ based, CPU-optimized inference engines (e.g., llama.cpp) that can leverage the CPU's Advanced Vector Extensions (AVX).

0.2. Optimal 500GB SSD Partitioning Strategy

The H.U.G.H. philosophy demands an "auditable" and "reliable" system. A default "guided" installation with a single root partition is insufficient. We must create a partition scheme that isolates the core OS, volatile system logs (for ingestion), and the H.U.G.H. system data itself. The use of Logical Volume Management (LVM) during the Kali Linux manual installation is required. This provides the flexibility to manage and resize volumes as the Knowledge Layer (the "Brain") grows. The following scheme isolates components for stability, backup, and auditable access.

Table 1: H.U.G.H. 500GB SSD LVM Partition Scheme

Partition	Mount Point	Filesystem	Size	Purpose
Physical 1	/boot/efi	vfat	512M	EFI System Partition (ESP) for boot.
Physical 2	/boot	ext4	1G	Houses kernel images and GRUB.
Physical 3	(none)	lvm2pv	~498.5G	LVM Physical Volume (to be

Partition	Mount Point	Filesystem	Size	Purpose
				carved up).
LV 1	/	ext4	100G	Kali Linux OS, tools, and applications.
LV 2	swap	swap	16G	Swap partition (sized for potential 8GB/16GB RAM).
LV 3	/var/log	ext4	20G	Isolates system logs for stable ingestion by GraphMERT.
LV 4	/opt/hugh	ext4	~362.5G	**The H.U.G.H. Core.** This volume will contain all models, the Neo4j database, and the Python virtual environment.

0.3. Kali Linux Base System Configuration

The Kali Linux operating system provides the environment, but it must be provisioned with the correct development libraries to build the H.U.G.H. stack from source, a necessity for the air-gap deployment.

A critical factor in this build is Kali's modern security posture. As of Kali 2024.4, the system enforces PEP 668, which explicitly forbids system-wide pip install commands to prevent conflicts with the apt package manager. The OS now *requires* the use of isolated Python virtual environments (venv). This OS-level constraint perfectly aligns with the H.U.G.H. philosophy of creating a self-contained, auditable, and isolated system.

The entire H.U.G.H. stack will be built and executed from a dedicated virtual environment located at /opt/hugh/venv.

Step-by-Step Provisioning:

1. Install Kali Linux from a live USB, selecting "Manual Partitioning" during installation.
2. Implement the LVM partition scheme detailed in Table 1.
3. Upon first boot, verify the desktop session is set to **X11 (Xorg)**, not Wayland. Talon voice control requires X11 for its screen/input capture APIs.
4. Open a terminal and install all necessary build and runtime dependencies via apt. This includes C++ compilers, Python headers, audio libraries, and CPU-optimization libraries. Neo4j (Phase 2) also requires a Java runtime.
   ```
   sudo apt update
   sudo apt install build-essential python3-dev python3-venv
   python3-pip git \
   libasound2-dev portaudio19-dev libopenblas-dev openjdk-11-jdk
   ```
5. Create and take ownership of the H.U.G.H. Core directory:
   ```
   sudo mkdir /opt/hugh
   sudo chown $USER:$USER /opt/hugh
   ```

6. Create the master Python virtual environment for the project :
   `python3 -m venv /opt/hugh/venv`

0.4. VS Code Setup & Coding Agent Recommendation

The development environment (VS Code) will be used in two modes: "Online" (during build, with internet access) and "Offline" (post-deployment, fully air-gapped). The recommended coding agent must support this transition.
Critical Recommendation: Microsoft AI Toolkit for VS Code.
This extension is the ideal choice as it functions as a "bridge" between the online build phase and the offline operational phase.

1. **Online Build Phase:** The developer will install the AI Toolkit extension. It can be configured to use their **OpenRouter Pro** access by setting it to a custom, OpenAI-compatible endpoint. This provides powerful, remote LLM assistance for assembling the stack.
2. **Offline Deployment Phase:** Once H.U.G.H. is deployed and its internal llama.cpp server is running (see Phase 1.2), the *same* AI Toolkit extension can be reconfigured. The developer can change the model source to "Foundry Local" or an ONNX model , filtering by "local on CPU". This allows the development environment to become a client of the very system it helped build, perfectly aligning with the "symbiotic" philosophy.

Phase 1: The Senses: Interface Layer Implementation

This phase builds the I/O layer, the "Senses" of the symbiote. It must be voice-first, locally processed, and bidirectional.

1.1. System-Wide Voice Control: Talon

Talon provides the system-wide "ears" and voice-command framework.

1. Download the Talon for Linux .deb package from the official website.
2. Install the package: sudo dpkg -i /path/to/talon_linux.deb.
3. Launch Talon. From the system tray icon, navigate to "Speech Recognition" and select **Conformer**. This will download and install the free, high-performance, local speech recognition engine.
4. All custom H.U.G.H. voice commands and Python glue scripts will be placed in the Talon user directory: ~/.talon/user/.

1.2. Local Inference Engine: Gemma 3 Nano (E2B) GGUF Deployment

This is the local "lightweight J.A.R.V.I.S." The specified model is **Gemma 3 Nano (E2B)** , an "Effective 2B" parameter model optimized for low-resource devices. We will use a quantized GGUF version (e.g., unsloth/gemma-3n-E2B-it-GGUF) and run it using the llama.cpp engine.
Build and Deployment Steps:

1. Activate the master virtual environment: source /opt/hugh/venv/bin/activate.
2. Clone the llama.cpp repository:

```
git clone https://github.com/ggerganov/llama.cpp.git
/opt/hugh/llama.cpp
```

3. Build the C++ core. The build flags are critical for CPU performance on the Ivy Bridge architecture. LLAMA_OPENBLAS=1 links the libopenblas-dev we installed , and LLAMA_AVX=1 enables the AVX instructions supported by the CPU.
   ```
   cd /opt/hugh/llama.cpp
   make LLAMA_OPENBLAS=1 LLAMA_AVX=1
   ```

4. Build the Python bindings, which are required for CrewAI (Phase 3) and other components:
   ```
   pip install -r requirements.txt
   pip install.
   ```

5. Launch the local, OpenAI-compatible API server. The model file (downloaded in Phase 5.2) will be placed in /opt/hugh/models/.
   ```
   /opt/hugh/llama.cpp/llama-server \
   -m /opt/hugh/models/gemma-3n-e2b.Q4_K_M.gguf \
   --host 127.0.0.1 --port 8080 \
   --n-ctx 4096 -ngl 0
   ```
   The -ngl 0 flag is mandatory. It sets n-gpu-layers to zero, forcing all 100% of the model layers to run on the CPU.

1.3. Local Speech Synthesis: Piper TTS

For the "voice" of H.U.G.H., a lightweight, fast, local Text-to-Speech (TTS) engine is required. **Piper TTS** is the definitive choice. It is explicitly designed for high-speed synthesis on low-power devices, including the Raspberry Pi.
Given that Piper achieves real-time synthesis on a Pi's ARM processor , it will run *significantly faster* than real-time on the 2012 Intel i5/i7. This confirms the TTS component will add negligible latency to the interaction loop.
Deployment:

1. Download the piper binary and a desired voice model (e.g., en_US-lessac-medium.onnx) from the official repository.
2. Place the binary and model in /opt/hugh/piper/ and /opt/hugh/models/, respectively.
3. Test synthesis from the command line:
   ```
   echo "H.U.G.H. is operational." | /opt/hugh/piper/piper --model \
   /opt/hugh/models/en_US-lessac-medium.onnx --output-raw | aplay \
   -r 22050 -f S16_LE -t raw -
   ```

1.4. The "Semiotic Glue": Bidirectional Python Scripting

This "glue" connects Talon (ears), Gemma 3n (local brain), and Piper (voice). This initial script provides the simple call-and-response loop. *Note: This script will be replaced in Phase 4 by the full agentic loop.*

1. **Talon Voice Command (~/.talon/user/hugh_interface.talon):** This file defines the voice command that triggers the Python script.

```
# This defines the "hugh" command, which captures all subsequent
speech
# as the 'text' variable.
hugh <user.text>:
    # This action executes a python script, passing the 'text'
    user.run_script("/opt/hugh/venv/bin/python",
"/opt/hugh/scripts/interface_glue.py", text)
```

2. **Python Glue Script (/opt/hugh/scripts/interface_glue.py):** This script (running in the venv) receives text from Talon, queries the llama.cpp server, and pipes the response to Piper TTS.

```
import sys
import requests
import subprocess

# 1. Get transcribed text from Talon (passed as argument)
text_from_talon = " ".join(sys.argv[1:])

# 2. Query Gemma 3n (Interface LLM) via the local server
try:
    response = requests.post(
        "http://127.0.0.1:8080/v1/chat/completions",
        headers={"Content-Type": "application/json"},
        json={
            "model": "gemma-3n-e2b",
            "messages": [{"role": "user", "content":
text_from_talon}]
        },
        timeout=30 # Set a 30-second timeout
    )
    llm_response_text =
response.json()['choices']['message']['content']
except requests.RequestException as e:
    llm_response_text = f"My interface layer is non-responsive.
Error: {e}"

# 3. Pipe LLM text to Piper TTS for audio output
try:
    piper_proc = subprocess.Popen(
       ,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE
    )
    # Use 'aplay' to play the raw audio stream
    aplay_proc = subprocess.Popen(
       ,
        stdin=piper_proc.stdout
    )
```

```
        # Write the text to Piper's stdin and close it
        piper_proc.stdin.write(llm_response_text.encode('utf-8'))
        piper_proc.stdin.close()
        piper_proc.wait()
        aplay_proc.wait()

    except Exception as e:
        print(f"Error in TTS pipeline: {e}")
```

1.5. Real-Time Performance Feasibility Analysis

A realistic performance expectation must be set. The 2012 dual-core Ivy Bridge CPU is severely resource-constrained. While llama.cpp is highly optimized, inference on a 2B parameter model (E2B) will be limited by CPU clock speed, core count, and memory bandwidth.
"Real-time" (i.e., sub-second) responses are **not possible**.
The expected latency for the full loop (Voice -> Talon -> Gemma -> Piper -> Ear) will be significant, likely in the **5-10 second** range per interaction. This is not a failure. This behavior is consistent with the "collaborative partner" philosophy—a thoughtful, considered pause—rather than the instantaneous reflex of a "servant" tool.

Phase 2: The Brain: Knowledge Layer Implementation

This phase constructs the persistent, auditable, and queryable "shared reality" for the user and the agentic swarm.

2.1. Local Graph Database: Neo4j Community Edition

The Cognee memory library requires a graph database backend to store its structured knowledge. Neo4j is the most mature, open-source, and developer-friendly option.

1. **Java Dependency:** The openjdk-11-jdk package installed in Phase 0.3 satisfies the Java runtime requirement.
2. **Add Neo4j Repository:** Add the official Neo4j apt repository. We will pin to version 4.4, a stable, long-term support (LTS) release.
   ```
   wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo
   apt-key add -
   echo "deb [trusted=yes] https://debian.neo4j.com stable 4.4" |
   sudo \
   tee /etc/apt/sources.list.d/neo4j.list
   sudo apt update
   ```
3. **Install Neo4j:**
   ```
   sudo apt install neo4j
   ```
4. **Configuration:** Edit /etc/neo4j/neo4j.conf to uncomment the following line. This binds the database to localhost, making it accessible to Cognee but not the outside network.

dbms.default_listen_address=127.0.0.1

5. **Start Service:** Launch the Neo4j service and set the initial admin password.
```
sudo neo4j start
# Set a secure password when prompted
neo4j-admin set-initial-password <your-secure-password>
```

2.2. AI Memory System: Cognee Library

Cognee is the AI memory library that unifies vector data and the knowledge graph. It will be configured to use our local llama.cpp server as its "brain" and Neo4j as its "memory."

1. **Installation:** Activate the venv and install Cognee:
```
source /opt/hugh/venv/bin/activate
pip install cognee
```

2. **Configuration:** Cognee is configured via environment variables. Create a .env file in the project root (/opt/hugh/.env). This configuration achieves the **Unified Inference Endpoint** architecture. Cognee's "Cognify" step (which extracts entities and relations) requires an LLM. We will point it to the *exact same* llama.cpp server used by the Interface layer.**Configuration File (/opt/hugh/.env):**
```
# --- Cognee LLM Configuration ---
# Use a 'custom' provider pointing to our local server
[span_69](start_span)[span_69](end_span)
LLM_PROVIDER="custom"
LLM_MODEL="gemma-3n-e2b"
LLM_ENDPOINT="http://127.0.0.1:8080"
LLM_API_KEY="dummy_key_not_required"

# --- Cognee Graph DB Configuration ---
# Override the default (Kuzu) to use Neo4j
GRAPH_PROVIDER=[span_63](start_span)[span_63](end_span)"neo4j"
GRAPH_DATABASE_URL="bolt://localhost:7687"
GRAPH_DATABASE_USERNAME="neo4j"
GRAPH_DATABASE_PASSWORD="<your_neo4j_password_from_step_2.1>"

# --- Cognee Vector DB Configuration ---
# Use the default local (LanceDB)
VECTOR_STORE_PROVIDER="lancedb"
VECTOR_STORE_PATH="/opt/hugh/data/vector_store"
```

2.3. The GraphMERT Pipeline (Part 1): Model Acquisition

The core of the "anti-Stark" philosophy is replacing a black-box core with an auditable neurosymbolic model. **GraphMERT** is this model: a compact 80M-parameter encoder that distills high-quality, reliable KGs from unstructured text. It has been shown to vastly outperform large LLMs (like 32B-parameter models) in generating factual, ontology-consistent KGs.
The provided materials describe the model but do not link to a public repository. The acquisition

of this model is a prerequisite and will be handled in Phase 5.2 (Air-Gap Download) using the user's Hugging Face Pro access. We will assume the model (e.g., princeton/graphmert-80m) is downloaded to /opt/hugh/models/graphmert and includes its necessary Python scripts (e.g., distill.py).

2.4. The GraphMERT Pipeline (Part 2): Seed KG Distillation

GraphMERT requires a "seed KG" to ensure its generated relations are "ontology-consistent". For the specified Kali Linux / cybersecurity domain, the canonical ontology is the **MITRE ATT&CK framework**. This framework is available in a machine-readable STIX 2.1 JSON format.

A Python script (/opt/hugh/scripts/build_seed_kg.py) must be created to generate this seed file:

1. Download the enterprise-attack.json file (see Phase 5.2).
2. Install the stix2 library: pip install stix2.
3. The script will parse the STIX 2.1 JSON, iterating through all STIX Domain Objects (SDOs) like attack-pattern (Techniques), intrusion-set (Groups), and malware.
4. It will then extract all STIX Relationship Objects (SROs) (e.g., uses, targets).
5. This graph of known-good cybersecurity relationships will be exported to a simple format (e.g., seed_kg.json). This file is the "seed" that primes GraphMERT's understanding of the domain.

2.5. The GraphMERT Pipeline (Part 3): Ingesting Unstructured Data

This is the primary distillation process, creating the H.U.G.H. Knowledge Graph.

1. **Input Data:** A directory of unstructured, domain-specific text. This includes the user's ~/Documents, saved Nmap/Burp reports, user notes, and—critically—the contents of the isolated /var/log partition.
2. **Ontology:** The seed_kg.json file generated in the previous step.
3. **Process:** The (assumed) GraphMERT distillation script is executed.
   ```
   source /opt/hugh/venv/bin/activate
   python /opt/hugh/models/graphmert/distill.py \
   --unstructured-data /path/to/user/docs,/var/log \
   --seed-ontology /opt/hugh/data/seed_kg.json \
   --output-graph /opt/hugh/data/hugh_kg_v1.json
   ```
4. **Output:** GraphMERT reads the input text and, using its neurosymbolic engine, generates new, reliable, ontology-consistent triples with provenance. For example, it will parse a log file and generate triples like (log_entry_45) --[contains_event]-- (failed_login_1) and (failed_login_1) --[targets_user]-- (root_user). The result is the final, auditable hugh_kg_v1.json.

2.6. The Cognee ECL Pipeline: Loading the "Brain"

With the reliable KG generated by GraphMERT, we now load it into Cognee's persistent memory (Neo4j) using its "Extract, Cognify, Load" (ECL) pipeline.

A Python script (/opt/hugh/scripts/load_brain.py) will perform this:

```
import cognee
import asyncio
```

```
import os
from dotenv import load_dotenv

async def load_hugh_memory():
    # Load the.env file from /opt/hugh/.env
    load_dotenv("/opt/hugh/.env")

    # 1. EXTRACT: Add the GraphMERT-generated KG file
    # This stages the file for processing.
    await cognee.add_file("/opt/hugh/data/hugh_kg_v1.json")

    # 2. COGNIFY & 3. LOAD:
    # This step is asynchronous.
    # Cognee reads the data, uses the local Gemma 3n LLM
    # (via the LLM_ENDPOINT in.env)
to[span_64](start_span)[span_64](end_span) understand and
    # process the entities, and automatically LOADS
    # the resulting graph into the Neo4j
database.[span_90](start_span)[span_90](end_span)
    await cognee.cognify()

    print("H.U.G.H. Knowledge Layer is cognified and loaded.")

if __name__ == "__main__":
    asyncio.run(load_hugh_memory())
```

Upon completion of this script, the "shared, auditable reality" is persistent and queryable.

Phase 3: The Will: Agentic Layer Implementation

This phase resolves the conflict with the proprietary "Pheromind" and deploys an open-source, air-gapped "swarm intelligence" framework.

3.1. Pheromind Conflict Resolution: Framework Analysis

The requirement is for an open-source, air-gap-compatible multi-agent framework. The top three Python-based candidates are AutoGen, LangGraph, and CrewAI.

Table 3: Agentic Framework Comparison (H.U.G.H. Context)

Framework	Core Philosophy	H.U.G.H. Fit & Rationale
AutoGen	**"Conversation-Driven"**. Flexible, emergent agent-to-agent chat. Good for research.	**Medium Fit.** The emergent, conversational nature is less ideal for the deterministic, auditable task execution required by an OS symbiote.
LangGraph	**"State-Machine"**. Provides full control over a graph-based workflow. Excellent for	**High (Technical) Fit.** This is a strong contender, aligning perfectly with the "auditable"

Framework	Core Philosophy	H.U.G.H. Fit & Rationale
	complex, auditable, and replayable logic.	requirement.
CrewAI	**"Role-Based"**. Models agents as a team with specific roles and goals. "Intuitive for team-like workflows".	**High (Philosophical & Technical) Fit.** This is the selected framework.

Selection: CrewAI
While LangGraph offers superior low-level control of the workflow state, **CrewAI**'s core "role-based" abstraction is a *direct semantic match* for the H.U.G.H. project's "semiotic relationship" and "partner" philosophy. We can define agents as collaborators (e.g., CybersecurityAnalyst, SystemExecutor, KnowledgeCurator) rather than just nodes in a graph. CrewAI is also production-ready and has first-class support for local LLMs via its LangChain backend.

3.2. Selected Framework: CrewAI Installation & Configuration

1. **Installation:** Activate the venv and install CrewAI and its associated tooling package:
   ```
   source /opt/hugh/venv/bin/activate
   pip install crewai crewai-tools
   ```

2. **Configuration:** CrewAI will be configured to use the same **Unified Inference Endpoint** (llama.cpp server) as the Interface and Knowledge layers. This is accomplished by defining a shared LLM object.Create a file: /opt/hugh/core/llm.py
   ```
   from langchain_openai import ChatOpenAI

   # This is the single, unified LLM definition for the entire
   H.U.G.H. system.
   # Any agent, tool, or process needing an LLM will import this
   function.

   def get_hugh_llm():
       """
       Returns an instance of the local LLM running on the
       llama.cpp server.
       """
       return ChatOpenAI(
           # Points to the server from Phase 1.2
           base_url="http://127.0.0.1:8080/v1",
           api_key="dummy_key_not_required",
           model_name="gemma-3n-e2b",
           temperature=0.7
       )
   ```

Phase 4: The Symbiosis: Full System Integration

This phase connects all three layers (Senses, Brain, Will) into a single, functional, symbiotic loop.

4.1. The H.U.G.H. System Architecture: Data Flow

The simple "glue" script from Phase 1.4 is now replaced by the full agentic workflow:

1. [User Voice] -> **(1) Talon:** Transcribes voice to text.
2. `` -> **(2) Talon Python Script:** Receives the text. It now imports the CrewAI components and calls Crew.kickoff(inputs={'task': text}).
3. `` -> **(3) CrewAI Agentic Layer:** A PlannerAgent receives the task. To "ground" its plan, it must query the "Brain."
4. [Query] -> **(4) KnowledgeTool:** The agent calls its KnowledgeGr[span_18](start_span)[span_18](end_span)aphQuery tool (see 4.3).
5. `` -> **(5) CrewAI Agentic Layer:** The agent receives the structured knowledge from Cognee/Neo4j. It forms a plan (e.g., "User asked to scan the local machine. I must run nmap.").
6. [Command] -> **(6) ShellTool:** The agent calls its ShellExecutor tool (see 4.2) with the argument command="nmap -sV 127.0.0.1".
7. `` -> **(7) CrewAI Agentic Layer:** The agent receives the stdout from the shell, updates the Knowledge Graph with the new findings (via another tool call), and formulates a final, natural-language response.
8. `` -> **(8) Python "Glue":** The Crew.kickoff() call returns the agent's final text response.
9. `` -> **(9) Piper TTS:** The script pipes this final text to the Piper binary.
10. [Audio Output] -> [User Ear].

4.2. Agent-OS Interaction: The "Shell" Custom Tool

The Agentic Layer needs "hands" to execute tasks on the OS. We will create a custom CrewAI tool. This tool will *not* use the CodeInterpre[span_47](start_span)[span_47](end_span)terTool , as that requires Docker, violating our lightweight, single-process philosophy. Instead, it uses Python's subprocess module directly.

Create file: /opt/hugh/core/tools/shell_tool.py

```
import subprocess
from crewai.tools import BaseTool
from pydantic import BaseModel, Field
from typing import Type

class ShellToolInput(BaseModel):
    """Input schema for the ShellExecutor tool."""
    command: str = Field(..., description="The shell command to
execute on Kali Linux.")

class ShellTool(BaseTool):
    name: str = "ShellExecutor"
    description: str = "Executes any shell command on the local Kali
Linux OS."
    args_schema: Type = ShellToolInput
```

```
    def _run(self, command: str) -> str:
        """Executes the shell command in a subprocess."""
        try:
            result = subprocess.run(
                command,
                shell=True,
                check=True,
                text=True,
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE,
                # Scope execution to the user's home directory
                cwd=os.path.expanduser("~")
            )
            return f"Command executed.
STDOUT:\n{result.stdout}\nSTDERR:\n{result.stderr}"
        except subprocess.CalledProcessError as e:
            return f"Error executing command. STDERR:\n{e.stderr}"
```

4.3. Agent-Brain Interaction: The "Cognee" Custom Tool

The Agentic Layer must query the "Brain" (Knowledge Layer) to build grounded plans. This tool, based on official Cognee examples , provides that access.
Create file: /opt/hugh/core/tools/knowledge_tool.py

```
import cognee
import asyncio
from crewai.tools import BaseTool
from pydantic import BaseModel, Field
from typing import Type

class KnowledgeInput(BaseModel):
    """Input schema for the KnowledgeGraphQuery tool."""
    query: str = Field(..., description="A natural language query for
the H.U.G.H. shared reality.")

class KnowledgeTool(BaseTool):
    name: str = "KnowledgeGraphQuery"
    description: str = "Queries the H.U.G.H. shared reality (knowledge
graph) for facts, entities, and relationships."
    args_schema: Type = KnowledgeInput

    def _run(self, query: str) -> str:
        """Runs the asynchronous Cognee search query."""
        try:
            # Cognee's search function is asynchronous
[span_114](start_span)[span_114](end_span)
            async def run_query():
                return await cognee.search(query_text=query)
```

```
        # Get or create an event loop to run the async function
        try:
            loop = asyncio.get_running_loop()
        except RuntimeError:
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)

        result = loop.run_until_complete(run_query())
        return str(result)

    except Exception as e:
        return f"Error querying knowledge graph: {e}"
```

4.4. Grounding the Semiotic Relationship: "Real-World Anchors"

The user's three "real-world anchor points" must be encoded at the root of the KG and be non-modifiable by the agent. Neo4j's IMMUTABLE keyword is an Enterprise-only feature and cannot be used in our Community Edition installation.
We will achieve "philosophical immutability" *programmatically* using Neo4j's role-based access control.

1. **Create Constraint and Anchor Node:** During the load_brain.py script, add the following Cypher execution to create a unique anchor node. MERGE makes this operation idempotent.
   ```
   // Creates a uniqueness constraint on the 'uuid' property for all
   'Anchor' nodes [span_119](start_span)[span_119](end_span)
   CREATE CONSTRAINT anchor_uuid IF NOT EXISTS FOR (a:Anchor) REQUIRE
   a.uuid IS UNIQUE;

   // Create the anchor node only if it doesn't exist
   MERGE (anchor:Anchor {uuid: "HUGH_ROOT_ANCHOR_v1"})
   ON CREATE SET
       anchor.point1 = "USER_PROVIDED_ANCHOR_1",
       anchor.point2 = "USER_PROVIDED_ANCHOR_2",
       anchor.point3 = "USER_PROVIDED_ANCHOR_3",
       anchor.description = "The immutable root anchor of the
   H.U.G.H. semiotic relationship."
   ```

2. **Create Agent-Specific Role:** Log into the Neo4j browser (http://localhost:7474) as the admin and create a new, less-privileged role for the H.U.G.H. agents.
   ```
   // Create the new role
   CREATE ROLE hugh_agent_role;

   // Grant read-only access to the entire graph
   GRANT MATCH ON GRAPH * TO hugh_agent_role;

   // Grant write/create/delete privileges for all nodes
   ```

```
GRANT CREATE, SET, REMOVE ON GRAPH * NODES * TO hugh_agent_role;

// *Crucially*, explicitly DENY write/delete privileges for nodes
with the 'Anchor' label
DENY SET, REMOVE ON GRAPH * NODES Anchor TO hugh_agent_role;
```

3. **Create Agent User and Configure Cognee:** Create a new user and assign it this role.
   ```
   CREATE USER hugh_agent SET PASSWORD '<agent-password>' CHANGE NOT
   REQUIRED;
   GRANT ROLE hugh_agent_role TO hugh_agent;
   ```

4. **Final Step:** Update the /opt/hugh/.env file to use these new, restricted credentials.
   ```
   GRAPH_DATABASE_USERNAME="hugh_agent"
   GRAPH_DATABASE_PASSWORD="<agent-password>"
   ```

This configuration ensures the agent swarm (connecting via Cognee) can read the anchors but is *programmatically forbidden* at the database level from ever altering or deleting them.

Phase 5: Deployment

This phase details the final "build and transfer" protocol to make the system fully air-gapped.

5.1. The Air-Gap Protocol (Part 1): Offline Python Environment

This process bundles all Python dependencies for offline installation.

1. **On an ONLINE Build Machine:**
   - Clone the H.U.G.H. project: git clone [your-project-repo] /tmp/hugh_build
   - Create a temporary venv: python3 -m venv /tmp/hugh_build/.venv
   - Activate: source /tmp/hugh_build/.venv/bin/activate
   - Install requirements: pip install -r /tmp/hugh_build/requirements.txt
   - Create the "wheelhouse" directory: mkdir /tmp/hugh_wheelhouse
   - Download all packages and dependencies:
     ```
     pip download -r /tmp/hugh_build/requirements.txt -d
     /tmp/hugh_wheelhouse
     ```
   - Copy the *entire* /tmp/hugh_wheelhouse directory to a USB drive.
2. **On the OFFLINE H.U.G.H. Machine (2012 MacBook):**
   - Mount the USB drive.
   - Activate the master venv: source /opt/hugh/venv/bin/activate
   - Install all packages from the wheelhouse, using --no-index to prevent any network access :
     ```
     pip install --no-index
     --find-links=/media/usb/hugh_wheelhouse \
     -r /opt/hugh/requirements.txt
     ```

5.2. The Air-Gap Protocol (Part 2): Model & Data Download

This process uses the user's HF Pro and OpenRouter access to download all necessary AI models and data *before* "pulling the plug." The huggingface-cli tool (installed via pip install huggingface_hub) is used.

Table 4: H.U.G.H. Air-Gap Download Manifest

Asset	Category	HF/Download Command (hf download or wget)
Gemma 3n E2B GGUF	Interface LLM	hf download unsloth/gemma-3n-E2B-it-GGUF --include "*.gguf" --local-dir /media/usb/hugh_models/gemma
GraphMERT 80M	Knowledge LLM	hf download <princeton/graphmert-80m> --local-dir /media/usb/hugh_models/graphmert
Piper TTS Voice	Interface TTS	hf download rhasspy/piper-voices --include "en_US-lessac-medium.*" --local-dir /media/usb/hugh_models/piper
MITRE ATT&CK STIX	Seed KG	wget https://github.com/mitre-attack/attack-stix-data/raw/master/enterprise-attack/enterprise-attack.json -P /media/usb/hugh_data/

After downloading to the USB, transfer all files from hugh_models to /opt/hugh/models/ and hugh_data to /opt/hugh/data/.

5.3. Final "Pulling the Plug" Checklist

1. [ ] Verify all Python packages are installed in /opt/hugh/venv.
2. [ ] Verify all models (.gguf, .onnx) are in /opt/hugh/models.
3. [ ] Verify all data (.json) is in /opt/hugh/data.
4. [ ] Run the load_brain.py script (Phase 2.6) to load the KG into Neo4j.
5. [ ] Run the Anchor grounding script (Phase 4.4) to secure the anchors.
6. [ ] Verify Neo4j service is running: sudo systemctl status neo4j.
7. [ ] Verify llama.cpp server is running in a screen or tmux session (Phase 1.2).
8. [ ] Verify Talon application is running.
9. [ ] **Action: Disconnect all network interfaces (Wi-Fi and Ethernet).**
10. The system is now fully air-gapped.

5.4. System Validation: The H.U.G.H. "Semiotic Test Suite"

To confirm all three layers are communicating correctly in the air-gapped environment, execute a final integration test. This test validates the full "semiotic loop" from tasking to knowledge retrieval to execution.

Create file: /opt/hugh/test_suite.py

```
import requests
import asyncio
import os
import cognee
from dotenv import load_dotenv
from core.llm import get_hugh_llm
from core.tools.shell_tool import ShellTool
from crewai import Agent, Task, Crew

# Load environment variables for Cognee
load_dotenv("/opt/hugh/.env")

def test_interface_layer():
    """Tests if the Gemma 3n (llama.cpp) server is responsive."""
    print("Testing Interface Layer (Gemma 3n)...")
    try:
        response = requests.post(
            "http://127.0.0.1:8080/v1/chat/completions",
            json={"model": "gemma-3n-e2b", "messages": [{"role":
"user", "content": "Ping"}]}
        )
        assert response.status_code == 200, "Server responded with
error"
        print(" Interface Layer is responsive.\n")
    except Exception as e:
        print(f"[FAIL] Interface Layer: {e}\n")

async def test_knowledge_layer():
    """Tests if Cognee can query the Neo4j-backed Knowledge Graph for
the anchor."""
    print("Testing Knowledge Layer (Cognee + Neo4j)...")
    try:
        result = await cognee.search("Describe the root anchor")
        assert "HUGH_ROOT_ANCHOR_v1" in str(result), "Anchor node not
found"
        print(" Knowledge Layer query successful.\n")
    except Exception as e:
        print(f"[FAIL] Knowledge Layer: {e}\n")

def test_agentic_layer():
    """Tests if a CrewAI agent can use the ShellTool to execute a
command."""
    print("Testing Agentic Layer (CrewAI + ShellTool)...")
    try:
```

```
        llm = get_hugh_llm()
        shell_tool = ShellTool()
        agent = Agent(role="Tester", goal="Test shell", llm=llm,
tools=[shell_tool], verbose=False)
        # Use a simple, non-destructive command
        task = Task(description="Echo 'Hello H.U.G.H.' to the
terminal.", agent=agent)
        crew = Crew(agents=[agent], tasks=[task], verbose=False)
        result = crew.kickoff()

        assert "Hello H.U.G.H." in result, "ShellTool did not return
expected output"
        print(" Agentic Layer executed shell command.\n")
    except Exception as e:
        print(f"[FAIL] Agentic Layer: {e}\n")

if __name__ == "__main__":
    print("--- H.U.G.H. SEMIOTIC VALIDATION SUITE ---")
    test_interface_layer()
    asyncio.run(test_knowledge_layer())
    test_agentic_layer()
    print("")
    print("The three layers are communicating. The system is
operational.")
```

To Validate: source /opt/hugh/venv/bin/activate python /opt/hugh/test_suite.py
If all tests pass, the semiotic relationship is established. Project H.U.G.H. is fully deployed.

Works cited

1. Apple MacBook Pro "Core i5" 2.5 13" Mid-2012 Specs - EveryMac.com, https://everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i5-2.5-13-mid-2012-unibody-usb3-specs.html 2. MacBook Pro (13-inch, Mid 2012) - Technical Specifications - Apple Support, https://support.apple.com/en-us/111958 3. MacBook Pro (Retina, 13-inch, Late 2012) - Technical Specifications - Apple Support, https://support.apple.com/en-us/118463 4. MacBook Pro (13-inch, Mid 2012) Eligible … - Apple Community, https://discussions.apple.com/thread/251421239 5. MAC CUDA driver fully compatible with macOS High Sierra 10.13 (error - Page 5, https://forums.developer.nvidia.com/t/mac-cuda-driver-fully-compatible-with-macos-high-sierra-10-13-error/54788?page=5 6. Run LLMs on Intel® GPUs Using llama.cpp, https://www.intel.com/content/www/us/en/developer/articles/technical/run-llms-on-gpus-using-llama-cpp.html 7. LLaMA Now Goes Faster on CPUs, https://justine.lol/matmul/ 8. Partition your drive in Kali Linux #KaliLinux #LinuxTutorial #LinuxTips - YouTube, https://www.youtube.com/watch?v=4iJUamPSyK0 9. How to install Kali Linux 2020.2 with manual disk partitioning and encryption - YouTube, https://www.youtube.com/watch?v=8N9jKWm-cKY 10. Installing Python Applications via pipx | Kali Linux Documentation, https://www.kali.org/docs/general-use/python3-external-packages/ 11. Talon 0.4.0 documentation, https://talonvoice.com/docs/ 12. Install Dependencies - Python

Developer's Guide, https://devguide.python.org/contrib/workflows/install-dependencies/ 13. Python bindings for llama.cpp - GitHub, https://github.com/abetlen/llama-cpp-python 14. Installing Neo4j — ClinicalKnowledgeGraph 1.0 documentation - Read the Docs, https://ckg.readthedocs.io/en/latest/intro/getting-started-with-neo4j.html 15. Neo4j Installation. I started programming about two years… | by Josh-T | The KickStarter, https://medium.com/the-kickstarter/neo4j-installation-e9724a1644f 16. Install packages in a virtual environment using pip and venv, https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/ 17. Deploying Python projects to air-gapped systems - DEV Community, https://dev.to/borisuu/deploying-python-projects-to-air-gapped-systems-2agm 18. AI Toolkit for Visual Studio Code, https://code.visualstudio.com/docs/intelligentapps/overview 19. Get started with AI Toolkit for Visual Studio Code - Microsoft Learn, https://learn.microsoft.com/en-us/windows/ai/toolkit/toolkit-getting-started 20. Explore models in AI Toolkit - Visual Studio Code, https://code.visualstudio.com/docs/intelligentapps/models 21. Using local AI/LLM in VS Code without third party software, https://peterfalkingham.com/2025/10/27/using-local-ai-llm-in-vs-code-without-third-party-software/ 22. Talon Voice, https://talonvoice.com/ 23. Installing Talon On A Linux Machine - YouTube, https://www.youtube.com/watch?v=YcMMSYVG8eY 24. unsloth/gemma-3n-E2B-it-GGUF - Hugging Face, https://huggingface.co/unsloth/gemma-3n-E2B-it-GGUF 25. Gemma 3n model overview | Google AI for Developers, https://ai.google.dev/gemma/docs/gemma-3n 26. unsloth/gemma-3n-E4B-it-GGUF - Hugging Face, https://huggingface.co/unsloth/gemma-3n-E4B-it-GGUF 27. unsloth/gemma-3n-E2B-it - Hugging Face, https://huggingface.co/unsloth/gemma-3n-E2B-it 28. ggml-org/llama.cpp: LLM inference in C/C++ - GitHub, https://github.com/ggml-org/llama.cpp 29. Best CPU TTS that can run on something like a raspberry pi is Piper. It can do r... | Hacker News, https://news.ycombinator.com/item?id=44382292 30. Really Fast TTS for Low-Performance Devices? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1imsoyk/really_fast_tts_for_lowperformance_devices/ 31. CPU speed comparison among KittenTTS, Piper, MatchaTTS, and Kokoro TTS #40 - GitHub, https://github.com/KittenML/KittenTTS/issues/40 32. Improve whisper performance on intel hardware - Home Assistant Community, https://community.home-assistant.io/t/improve-whisper-performance-on-intel-hardware/699427 33. Piper Voice Samples, https://rhasspy.github.io/piper-samples/ 34. rhasspy/piper: A fast, local neural text to speech system - GitHub, https://github.com/rhasspy/piper 35. System Level Python Macros With Voice Commands Using Talon - bekk.christmas, https://www.bekk.christmas/post/2021/22/system-level-python-macros-with-voice-commands-using-talon 36. how to pipe whisper.cpp transcripts into sdin · Issue #2161 · ggml-org/llama.cpp - GitHub, https://github.com/ggerganov/llama.cpp/issues/2161 37. Intel Core i5-3470 Benchmarks - Geekbench Browser, https://browser.geekbench.com/processors/intel-core-i5-3470 38. Which specs have the biggest performance impact for CPU GGUF inference? - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1dtxmxb/which_specs_have_the_biggest_performance_impact/ 39. Build a Knowledge Graph from a Python Repo: A Simple Guide - Cognee, https://www.cognee.ai/blog/deep-dives/repo-to-knowledge-graph 40. Memory Fragment Projection: Personalized Knowledge Graph Layer - Cognee, https://www.cognee.ai/blog/deep-dives/memory-fragment-projection-from-graph-databases 41. topoteretes/cognee: Memory for AI Agents in 6 lines of code - GitHub, https://github.com/topoteretes/cognee 42. Graph Databases Explained: Better Way to Represent Connections - Cognee,

https://www.cognee.ai/blog/fundamentals/graph-databases-explained 43. Wrote a plain-English explainer on graph DB fundamentals (with a lot of Neo4j love) - Reddit, https://www.reddit.com/r/Neo4j/comments/1kssnx5/wrote_a_plainenglish_explainer_on_graph_db/ 44. Install Java on Kali Linux - Shouts.dev, https://shouts.dev/articles/install-java-on-kali-linux 45. command to install java jdk latest version on kali - Unix & Linux Stack Exchange, https://unix.stackexchange.com/questions/621896/command-to-install-java-jdk-latest-version-on-kali 46. Kali Neo4j BloodHound setup!!!. Take a snapshot before attempting this… | by HacktheBoxWalkthroughs | Medium, https://medium.com/@Q2hpY2tlblB3bnk/neo4j-bloodhound-setup-a2b0f866c4cc 47. DuckDB × cognee: Run SQL Analytics Right Beside Your Graph-Native RAG - MotherDuck, https://motherduck.com/blog/duckdb-cognee-sql-analytics-graph-rag/ 48. The Ultimate AI Engineer's Guide to the Official Cognee MCP Server - Skywork.ai, https://skywork.ai/skypage/en/ultimate-ai-engineer-guide-cognee-mcp-server/1977912822261551104 49. LLM Providers - Cognee Documentation, https://docs.cognee.ai/setup-configuration/llm-providers 50. [2510.09580] GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - arXiv, https://arxiv.org/abs/2510.09580 51. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data | alphaXiv, https://www.alphaxiv.org/overview/2510.09580v1 52. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data | Cool Papers, https://papers.cool/arxiv/2510.09580 53. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - Paper Detail - Deep Learning Monitor, https://deeplearn.org/arxiv/643479/graphmert:-efficient-and-scalable-distillation-of-reliable-knowledge-graphs-from-unstructured-data 54. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - Semantic Scholar, https://www.semanticscholar.org/paper/37245abf2595b9eab44c94eaa42da657f6ce108b 55. (PDF) GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - ResearchGate, https://www.researchgate.net/publication/396457862_GraphMERT_Efficient_and_Scalable_Distillation_of_Reliable_Knowledge_Graphs_from_Unstructured_Data 56. ATT&CK Data & Tools, https://attack.mitre.org/resources/attack-data-and-tools/ 57. Cyber Threat Modelling by leveraging an open source attack graph and activity thread graph tool - IdoubleS, https://idoubles.net/resources/cti-stix-diamond-activity-attack-graph.pdf 58. cognee - Scalable Data Layer for AI Apps - ECL Pipelines - dltHub, https://dlthub.com/blog/cognee 59. Build faster AI memory with Cognee & Redis, https://redis.io/blog/build-faster-ai-memory-with-cognee-and-redis/ 60. CrewAI vs LangGraph vs AutoGen: Choosing the Right Multi-Agent AI Framework, https://www.datacamp.com/tutorial/crewai-vs-langgraph-vs-autogen 61. Top 5 Open-Source Agentic Frameworks - Research AIMultiple, https://research.aimultiple.com/agentic-frameworks/ 62. LangGraph vs AutoGen vs CrewAI: Best Multi-Agent Tool? - Amplework, https://www.amplework.com/blog/langgraph-vs-autogen-vs-crewai-multi-agent-framework/ 63. Comparing 4 Agentic Frameworks: LangGraph, CrewAI, AutoGen, and Strands Agents | by Dr Alexandra Posoldova | Medium, https://medium.com/@a.posoldova/comparing-4-agentic-frameworks-langgraph-crewai-autogen-and-strands-agents-b2d482691311 64. Building AI Agents with CrewAI using Python - Aman Kharwal, https://amanxai.com/2025/07/01/building-ai-agents-with-crewai-using-python/ 65. How to run Crew.ai with local llms (Ollama) | by Mihir - Medium,

https://medium.com/@Mihir8321/how-to-run-crew-ai-with-local-llms-ollama-e11453b25cde 66. Create Custom Tools - CrewAI Documentation, https://docs.crewai.com/en/learn/create-custom-tools 67. Tools - CrewAI Documentation, https://docs.crewai.com/en/concepts/tools 68. Code Interpreter - CrewAI Documentation, https://docs.crewai.com/en/tools/ai-ml/codeinterpretertool 69. Using AI Agents to Execute Shell Scripts with Langgraph using ollama: A Smarter Approach to Automation | by ETL , ELT , Data And AI/ML Guy | Medium, https://medium.com/@Shamimw/using-ai-agents-to-execute-shell-scripts-with-langgraph-using-ollama-a-smarter-approach-to-679fd3454b09 70. Call a Python Script - Crews - CrewAI, https://community.crewai.com/t/call-a-python-script/2092 71. CrewAI Add Memory with Cognee: AI Agents with Semantic Memory, https://www.cognee.ai/blog/deep-dives/crewai-memory-with-cognee 72. Cypher Cheat Sheet - Neo4j, https://neo4j.com/docs/cypher-cheat-sheet/current/ 73. Database privileges - Operations Manual - Neo4j, https://neo4j.com/docs/operations-manual/current/authentication-authorization/database-administration/ 74. Constraints - Cypher Manual - Neo4j, https://neo4j.com/docs/cypher-manual/4.2/administration/constraints/ 75. How to make a node property immutable in neo4j? - Stack Overflow, https://stackoverflow.com/questions/48404019/how-to-make-a-node-property-immutable-in-neo4j 76. How to install packages offline? - python - Stack Overflow, https://stackoverflow.com/questions/11091623/how-to-install-packages-offline 77. Running python on air-gapped systems - Reddit, https://www.reddit.com/r/Python/comments/1961v23/running_python_on_airgapped_systems/ 78. How to create virtual env with Python 3? - Stack Overflow, https://stackoverflow.com/questions/43069780/how-to-create-virtual-env-with-python-3 79. How to install python packages with all dependencies offline via pip3? - Super User, https://superuser.com/questions/1523218/how-to-install-python-packages-with-all-dependencies-offline-via-pip3 80. How To Download and Use a Hugging Face Model Locally | Full Stack Wizardry - Medium, https://medium.com/full-stack-engineer/how-to-download-and-use-a-hugging-face-model-0e5f2bbc8682 81. Test Suites - Vapi Docs, https://docs.vapi.ai/test/test-suites 82. How to Evaluate Voice AI Agents: A Practical, End-to-End Framework for Quality, Reliability, and Speed - DEV Community, https://dev.to/kuldeep_paul/how-to-evaluate-voice-ai-agents-a-practical-end-to-end-framework-for-quality-reliability-and-k44