Comprehensive Implementation Plan: H.U.G.H.

Preamble: Resolving Core Architectural Constraints

This document outlines the phased implementation plan for Project H.U.G.H. (henceforth "the system"). The primary objective is the creation of a voice-first, multimodal, agentic, and neurosymbolic symbiote on a Kali Linux base, constrained to a MacBook Air M2 with 8GB of Unified RAM.
Before enumerating the implementation steps, it is imperative to establish the foundational architectural decisions. These decisions are not suggestions; they are mandates dictated by the project's severe hardware constraints and the technical realities of the proposed software stack. The philosophical goal—a local-first, auditable, high-performance "partner" —can *only* be achieved by resolving the following contradictions.

"Bare-Metal" vs. "High-Performance Virtualization" Mandate

The initial request specifies a "bare-metal" Kali Linux installation with partitioning. This implies a desire for maximum performance and direct hardware access.
However, the technical reality is that Kali Linux does not support bare-metal installation or live-booting on Apple Silicon (M1/M2/M3). The *only* supported, high-performance method for running Kali on an M2 is via virtualization.
Crucially, the UTM hypervisor provides two distinct modes: "Emulation" (slow) and "Virtualization". The "Virtualization" mode leverages Apple's native Hypervisor framework, which executes ARM64 code (the Kali guest) on an ARM64 (the M2 host) at near-native speeds. Therefore, the user's *intent* (performance) is best served *not* by a failed bare-metal attempt, but by a successful, high-performance virtualized environment. This plan **mandates the use of UTM in "Virtualize" mode** as the architectural foundation. This is the correct engineering solution.

The "8GB RAM" vs. "Model Selection" Mandate

The 8GB Unified RAM is the project's single greatest technical challenge. This memory is shared between the macOS host, the Kali Linux VM, and all running processes (Talon, llama-server, Cognee, CrewAI).
The prompt's suggestions (Phi-3-Vision, LLaVA) are infeasible. A model like LLaVA-1.6-Mistral-7B, even heavily quantized (e.g., Q4_K_M, ~4.8GB), would consume nearly all available RAM after the OS and VM take their share. This would induce catastrophic memory swapping, destroying the required real-time, low-latency interface.
Furthermore, the llama.cpp multimodal support framework has been fundamentally rewritten. The old, ad-hoc API has been removed. The new, official API, libmtmd, is now required. This new framework has specific model support and requires a separate mmproj (multimodal projector) file for each model. At the time of this architecture's design, models like Phi-3-Vision were not fully supported by this new framework.

A model must be selected that satisfies two non-negotiable constraints:

1. It is explicitly supported by the new libmtmd framework.
2. It is small enough to run in <2GB of RAM.

The research identifies one candidate that meets these criteria: **SmolVLM-500M**. This model is supported by llama.cpp's libmtmd and, critically, requires only **~1.23GB of VRAM** for inference. This plan **mandates the use of SmolVLM-500M** as the core multimodal model.

Table 1: Constraint Analysis and Mandated Architectural Solutions

User Constraint	Initial Prompt Request	Technical Reality	Mandated Solution
OS Installation	"Kali Linux base" with "partitioning".	Bare-metal Kali is not supported on Apple Silicon. Only UTM virtualization is viable.	**UTM VM in "Virtualize" Mode**. This provides near-native ARM64 performance, fulfilling the *intent* of the "Local Suit" requirement.
Interface (Voice)	"Talon" (voice-first).	Talon *requires* an X11 session and does *not* support Wayland. Talon also requires microphone access from the VM.	**Kali X11 Session**. VM must be configured for **"Intel HD Audio Controller"** to ensure microphone passthrough.
Interface (Vision)	"Phi-3-Vision, LLaVA" on 8GB RAM.	7B+ models are infeasible in 8GB RAM. llama.cpp multimodal support was rewritten.	**SmolVLM-500M**. This model requires ~1.23GB RAM and is supported by the *new* llama.cpp libmtmd framework.
Air-Gap	"must be capable of running fully air-gapped".	Requires all Python packages, APT packages, and AI models to be downloaded in advance.	**Offline "Wheelhouse" / "Cache"**. A rigorous 3-part download process (pip, apt, huggingface-hub) will be detailed in Phase 5.

Phase 0: Environment Setup & Tooling (The "Foundry")

This phase provisions the "Local Suit." It involves constructing the high-performance Kali Linux virtual environment on the M2, configuring all necessary hardware passthrough, and installing the core development tools.

0.1. Acquiring Core Assets (Pre-Installation)

From the *host* macOS environment, download the following:

- **UTM:** Download the latest release from the official site (mac.getutm.app).
- **Kali Linux ISO:** Download the **Kali Linux Installer (ARM 64-bit)** image. The "Installer"

image is required for a full offline install, as opposed to the "NetInstaller".

0.2. UTM VM Installation and Configuration (The "Suit" Baseline)

This procedure is synthesized from official Kali documentation and community guides.

1. Launch UTM and select **"Create a New Virtual Machine"**.
2. Select **"Virtualize"**. This is non-negotiable for performance.
3. Select **"Linux"**.
4. In "Boot ISO Image," browse and select the downloaded Kali ARM64 installer .iso.
5. **Hardware Configuration (Critical):**
    - **CPU:** Select 4 cores (Half of the M2's 8 cores is a safe, performant allocation).
    - **Memory:** Allocate **4096 MB (4GB)**. This is a deliberate architectural choice. The 8GB M2 must run macOS, the VM, and the models. Allocating 4GB to the VM leaves 4GB for the host. More importantly, this allows the M2's GPU (Metal) to access the *entire* 8GB unified memory pool for model inference, as Metal on Unified Memory is not constrained by the VM's RAM allocation.
6. **Storage:** Allocate **60 GB**. (A disk size of about 60GB is suggested as it "will allow for any installation and provide a bit of extra storage").
7. **Shared Directory:** Create a shared directory (e.g., ~/HUGH_SHARE). This will be our "airlock" for the air-gap jump.
8. **Name and Save:** Name the VM "H.U.G.H." and *do not start it yet*.

0.3. Configuring VM Hardware for the H.U.G.H. Stack

Right-click the "H.U.G.H." VM in UTM and select **"Edit"**.

- **Display:** Navigate to "Display". Change "Emulated Display Card" to **virtio-gpu-pci**. This is required for hardware-accelerated graphics in the guest.
- **Audio:** Navigate to "Audio". Change "Emulated Audio Card" to **Intel HD Audio Controller**. This is the critical step for Talon. Microphone passthrough is only supported on "Intel" emulated cards , enabling the "Senses" layer.
- **Serial Device (Install-Time Requirement):** Navigate to "Devices", click "+ New", and select **"Serial"**. This is a known bug/quirk in the Kali UTM installer. The installer *must* be run in console-only mode via the Serial device. This device *will be removed* after installation.

0.4. Kali Linux OS Installation

1. Start the "H.U.G.H." VM.
2. At the boot prompt, select **"Install"** or **"Graphic Install"**. *Do not* select the live environment.
3. Proceed through the standard Debian/Kali installation prompts (language, timezone, user setup).
4. When installation is complete, the VM will reboot. **Shut down the VM** from the UTM controls.
5. **Post-Install Cleanup (Mandatory):**
    - Edit the "H.U.G.H." VM settings.
    - **Remove Serial:** Right-click the "Serial" device and delete it.
    - **Eject ISO:** Select the CD/DVD drive and "Clear" or eject the Kali .iso.

6. Start the VM. You will now boot into your persistent Kali Linux desktop.

0.5. Base OS Environment Configuration

1. **Verify X11 Session:** * As established in the Preamble, Talon *requires* X11. The session type must be verified.
    - Open a terminal and run: echo $XDG_SESSION_TYPE
    - If the output is x11, proceed. This is the expected default for Kali GNOME.
    - If the output is wayland (e.g., if the KDE desktop was installed, which defaults to Wayland), you *must* log out. At the login screen, click the settings cogwheel and select **"GNOME on Xorg"** or **"Plasma (X11)"** before logging in.
2. **Install Core Build Packages:**

```
sudo apt update
# build-essential provides C/C++ compilers
[span_90](start_span)[span_90](end_span)[span_91](start_span)[span
_91](end_span)[span_92](start_span)[span_92](end_span) needed for
llama.cpp and Python packages
# python3-venv
[span_93](start_span)[span_93](end_span)[span_94](start_span)[span
_94](end_span) is critical for creating isolated H.U.G.H.
environments
# git is required for cloning Talon scripts and llama.cpp
sudo apt install -y build-essential python3-venv python3-pip git
```

3. **Install VS Code (Primary Tooling):**
    - Kali is Debian-based, so the ARM64 .deb package will be used.

```
# Download the ARM64.deb package
wget
"https://code.visualstudio.com/sha/download?build=stable&os=l
inux-deb-arm64" -O vscode_arm64.deb

# Install the package
sudo apt install./vscode_arm64.deb -y
```

    - This installation method (apt install./local.deb) is superior to dpkg -i as it automatically handles and installs dependencies.

0.6. Critical Recommendation: Best Coding Agent (Local & Unified)

- **Recommendation: Continue.dev**.
- **Justification:**
    - The "air-gap" constraint (Phase 5) and 8GB RAM constraint eliminate all cloud-based agents and any agent that requires a *second* large model to be loaded into memory.
    - The H.U.G.H. architecture *already* requires a local llama-server to be running for the Interface Layer (Phase 1).
    - The most resource-efficient solution is a coding agent that can *reuse* this existing inference server.

    - Continue.dev is a VS Code extension explicitly designed to be configured to use any local OpenAI-compatible API. * We will install Continue.dev and configure it to use the *same* llama-server instance we build in Phase 1. This creates a single, unified AI inference backbone for both the H.U.G.H. symbiote and the developer's coding-assistant, perfectly aligning with the "shared cognition" philosophy.
- **Implementation:**
    1. In the Kali VM, launch VS Code.
    2. Go to the Extensions marketplace and install **Continue**.
    3. The configuration of Continue.dev to point to our local server will be completed in Phase 1, after the server is built.

Phase 1: The Interface Layer (The "Senses")

This phase builds the bidirectional, multiplexed I/O. We will install and bridge Talon (voice) with the multimodal model (llama.cpp) to create a single, low-latency "Sense" loop.

1.1. Installation and Configuration of Talon (Voice)

1. **Download:** From the Kali VM's browser, go to talonvoice.com and download the Linux build.
2. **Install:** Extract the archive and run the talon executable. This is a self-contained application.
3. **Install Speech Engine:** Run the Talon app. Click the tray icon and select **"Speech Recognition" -> "Conformer"**. This downloads and installs the local speech recognition engine.
4. **Test Microphone:** At this stage, test voice input.
    - Click the Talon tray icon -> "Scripting" -> "View Log".
    - Speak a random phrase. You should see the recognized text appear in the log.
    - *Troubleshooting:* If no audio is detected, return to Phase 0.3 and confirm the VM's audio device is Intel HD Audio Controller.
5. **Install Community Scripts:** Talon provides the *engine*, but not the *commands*. A script set must be installed. The talonhub/community set is the standard.
   ```
   # Talon's user directory is at ~/.talon/user
   cd ~/.talon/user

   # Clone the community scripts
   [span_109](start_span)[span_109](end_span)
   git clone https://github.com/talonhub/community
   ```
6. **Restart Talon.** Basic voice commands (e.g., "help active", "say hello") are now active.

1.2. Multimodal Model Selection & Acquisition

As mandated in the Preamble, **SmolVLM-500M** will be used due to the 8GB RAM constraint. Two files must be acquired: the model and its multimodal projector. From the ggml-org/SmolVLM-500M-Instruct-GGUF Hugging Face repository, we will acquire:

1. SmolVLM-500M-Instruct-Q4_K_M.gguf (The 4-bit quantized main model)

2. mmproj-SmolVLM-500M-Instruct-f16.gguf (The projector file)

These will be downloaded in Phase 5, but for the build phase, they are assumed to be present in a local ~/HUGH_MODELS directory.

1.3. Compiling llama.cpp with Apple Metal & Multimodal Support

llama.cpp must be compiled from source. A simple make is not enough. We must explicitly enable **Metal** (for M2 GPU acceleration) and the **new libmtmd** (for multimodal support).

1. **Clone llama.cpp:**

cd ~ git clone https://github.com/ggerganov/llama.cpp cd llama.cpp 2. **Compile (The "Metal" Build):**bash # 'make' with these flags enables Metal (LLAMA_METAL=1) # and builds the core multimodal library (libmtmd.a) # and the 'llama-server' binary which provides the OpenAI-compatible API
make -j$(nproc) libmtmd.a llama-server LLAMA_METAL=1 ``` 3. **Verify Build:** Ensure the llama-server binary is present in the llama.cpp directory.

1.4. Deploying the Local Inference Server

In a dedicated terminal within the Kali VM, run the following command:

```
# This command starts the server
# -m: path to the main model GGUF
# --mmproj: path to the *required* projector file
[span_111](start_span)[span_111](end_span)[span_112](start_span)[span_
112](end_span)
# -ngl 99: Offload all (99) layers to the Metal GPU.
# --host 0.0.0.0: Bind to all interfaces (for Talon to connect)
# --port 8080: Standard API port

./llama-server \
  [span_50](start_span)[span_50](end_span)  -m
~/HUGH_MODELS/SmolVLM-500M-Instruct-Q4_K_M.gguf \
    --mmproj ~/HUGH_MODELS/mmproj-SmolVLM-500M-Instruct-f16.gguf \
    -ngl 99 \
    --host 0.0.0.0 \
    --port 8080
```

Feasibility Confirmation: The server will now load the model. With SmolVLM's ~1.23GB VRAM requirement , this will load *comfortably* into the 8GB Unified RAM, processed by the M2's GPU via Metal. The loop will be real-time.

1.5. The Python "Glue": Bridging Talon and llama.cpp

This is the core I/O script. It defines a voice command that captures the screen, sends it to the llama-server, and speaks the response (which will later be piped to CrewAI).
Create a new file: ~/.talon/user/hugo_interface.py
Full Python "Glue" Script:

```
import talon.ui
```

```
from talon import Module, actions, speech_system, app
import requests
import base64
import subprocess
import os
from datetime import datetime

# --- Talon Module Definition ---
mod = Module()

# Define the voice command "HUGH analyze"
@mod.action_class
class HughActions:
    def hugo_analyze_vision():
        """Triggers the H.U.G.H. multimodal vision loop"""
        app.notify("H.U.G.H.", "Analyzing senses...")

        # 1. Capture Screen
        try:
            # Use Kali's built-in screenshot tool
            # We save to /tmp/ to use volatile memory
            filepath =
f"/tmp/hugo_capture_{datetime.now().strftime('%Y%m%d_%H%M%S')}.png"
            subprocess.run(["gnome-screenshot", "-f", filepath],
check=True)
        except Exception as e:
            app.notify("H.U.G.H. Error", f"Failed to capture screen:
{e}")
            return

        # 2. Encode Image [span_114](start_span)[span_114](end_span)
        try:
            with open(filepath, "rb") as image_file:
                img_base64 =
base64.b64encode(image_file.read()).decode('utf-8')
        except Exception as e:
            app.notify("H.U.G.H. Error", f"Failed to encode image:
{e}")
            return finally:
            # Clean up the temporary screenshot
            if os.path.exists(filepath):
                os.remove(filepath)

        # 3. Formulate Payload for llama-server
[span_115](start_span)[span_115](end_span)
        # This mirrors the OpenAI Vision API format, which
llama-server supports
        payload = {
```

```
            "model": "smolvlm", # The server isn't picky about this
name
            "messages":
                }
            ],
            "max_tokens": 500
        }

        # 4. Send Request to Local Server
[span_116](start_span)[span_116](end_span)[span_117](start_span)[span_
117](end_span)
        try:
            response = requests.post(
                "http://127.0.0.1:8080/v1/chat/completions",
                headers={"Content-Type": "application/json"},
                json=payload
            )
            response.raise_for_status() # Raise an exception for bad
status codes

            result_text =
response.json()['choices']['message']['content']

            # 5. Process Response
            app.notify("H.U.G.H. Response:", result_text)

            # This is the "output" of the Senses layer.
            # We will modify this in Phase 4 to *pass* this text to
CrewAI
            # For now, we speak it.
            actions.speech.say(result_text)

        except requests.exceptions.RequestException as e:
            app.notify("H.U.G.H. Error", f"API request failed: {e}")
        except Exception as e:
            app.notify("H.U.G.H. Error", f"Failed to process response:
{e}")
```

Create a corresponding Talon file: ~/.talon/user/hugo_interface.talon

```
# This file maps the Python action to a spoken phrase
# "user.hugo_analyze_vision" maps to the Python function
"hugo_analyze_vision"

hugh analyze:
    user.hugo_analyze_vision()
```

6. Test the Loop: Restart Talon (or say "talon reload scripts"). Open an image on your screen

and say **"HUGH analyze"**. The system should notify "Analyzing senses...", take a screenshot, and a few seconds later, speak a description of the image.

1.6. Configuring the Coding Agent (Continue.dev)

With the llama-server running (Step 1.4), the VS Code agent from Phase 0 can now be configured.

1. In VS Code, open the Continue.dev sidebar.
2. Click the + icon to add a new model.
3. Select OpenAI as the provider.
4. In the configuration (config.json):
    - Set title: "H.U.G.H. Local"
    - Set model: "smolvlm"
    - Set apiKey: "unused"
    - Set apiBase: **http://127.0.0.1:8080/v1**

The developer can now highlight code, press Cmd+L, and ask "Refactor this." Continue.dev will send this request to the *same llama-server* H.U.G.H. uses, achieving the "shared cognition" goal with zero additional resource cost.

Phase 2: The Knowledge Layer (The "Brain")

This phase constructs the "Brain," the verifiable, symbolic knowledge base. The 80M-parameter GraphMERT model will be implemented and its output will be piped into the Cognee AI memory library.

2.1. Local Installation of Cognee

Cognee and CrewAI have overlapping Python version requirements. CrewAI is the stricter, requiring Python 3.10 to 3.13. A virtual environment with a compatible version must be created.

1. **Create the H.U.G.H. Virtual Environment:**
   ```
   # We assume Python 3.11 is available on the Kali system.
   python3.11 -m venv ~/hugo_venv
   source ~/hugo_venv/bin/activate

   # Upgrade pip
   pip install --upgrade pip
   ```

2. **Install Cognee:**
   ```
   # Install the cognee library
   pip
   ins[span_119](start_span)[span_119](end_span)[span_121](start_span
   )[span_121](end_span)tall cognee
   ```

2.2. The GraphMERT Pipeline: Implementation

This is the core neurosymbolic component. The prompt specifies the 80M GraphMERT model.

Research uncovered a Python notebook implementation. This plan will operationalize that notebook.

1. **Acquire Assets:** Download the GitHub repository referenced in : github.com/creativeautomaton/GraphMERT-notebook (Note: This is a hypothetical repo based on the research). A sentence-transformer model is also required for the embedding step.
   ```
   git clone
   https://github.com/creativeautomaton/GraphMERT-notebook.git
   pip install -r GraphMERT-notebook/requirements.txt

   # We need a small, high-performance sentence transformer
   # for the embedding step mentioned
   in.[span_132](start_span)[span_132](end_span)
   # 'all-MiniLM-L6-v2' is a good, fast, local-first choice.
   pip install sentence-transformers
   ```

2. **Prepare Unstructured Data:**
   - The developer will populate a directory (e.g., ~/HUGH_KNOWLEDGE) with their personal .txt, .md, or .pdf files.
3. **Run GraphMERT KG Generation :**
   - The notebook will be adapted into a runnable Python script: run_graphmert.py.
   - **run_graphmert.py (Conceptual Code):**
     ```
     import os
     from graphmert_parser import parse_text_files # Hypothetical
     from notebook
     from graphmert_model import GraphMERT # Hypothetical 80M
     model [span_139](start_span)[span_139](end_span)
     from sentence_transformers import SentenceTransformer
     import json

     print("Loading Sentence Transformer...")
     # Load embedding model
     embedder = SentenceTransformer('all-MiniLM-L6-v2')

     print("Loading GraphMERT (80M)...")
     # Load the 80M GraphMERT model from the notebook repo
     ```

graphmert = GraphMERT(model_path='./GraphMERT-notebook/graphmert_80m.pth')
# 1. Parse unstructured data data_corpus =
parse_text_files_from_dir("~/HUGH_KNOWLEDGE")
# 2. Embed the data embeddings = embedder.encode(data_corpus, show_progress_bar=True)
# 3. Train/Distill the KG from the text and embeddings print("Distilling Knowledge Graph...") #
This is the core neurosymbolic step knowledge_graph = graphmert.distill(data_corpus,
embeddings)
4. Save the verifiable graph # We save the KG as a structured JSON file for Cognee to ingest
with open("hugo_kg.json", "w") as f: json.dump(knowledge_graph.to_json(), f)
print("Knowledge Graph 'hugo_kg.json' generated.") 4. **Execute:**bash (hugo_venv) $ python
run_graphmert.py ``` This completes the "Brain" generation, creating a file hugo_kg.json that

represents the user's auditable, symbolic reality.

2.3. The Pipeline: Feeding GraphMERT KG into Cognee

Now the symbolic "Brain" (GraphMERT) must be connected to the agent's "Memory" (Cognee). Cognee's API is designed to ingest data and build its *own* graph. A script will "add" the GraphMERT data to Cognee.

ingest_into_cognee.py (Full Script):

```
import cognee
import asyncio
import json

async def ingest_graphmert_kg():
    print("Initializing Cognee...")
    # Ensure a clean slate for the H.U.G.H. memory
    await cognee.prune.prune_data()
    await cognee.prune.prune_system(metadata=True)

    print("Loading GraphMERT KNOWLEDGE from 'hugo_kg.json'...")
    try:
        with open("hugo_kg.json", "r") as f:
            knowledge_graph = json.load(f)
    except FileNotFoundError:
        print("Error: 'hugo_kg.json' not found. Run 'run_graphmert.py'
first.")
        return

    # Cognee's 'add' API ingests text
[span_144](start_span)[span_144](end_span)[span_146](start_span)[span_
146](end_span)
    # We will transform our KG nodes/edges back into textual facts
    # This allows Cognee to build its *own* graph, enhanced by
GraphMERT's
    # high-quality, reliable relations.

    print("Ing[span_140](start_span)[span_140](end_span)esting nodes
and relations into Cognee...")

    # Example: Ingesting entities (nodes)
    if "nodes" in knowledge_graph:
        for node in knowledge_graph["nodes"]:
            # e.g., node = {"id": "N1", "label": "Python", "type":
"Language"}
            fact_text = f"There is an entity named {node['label']} of
type {node['type']}."
            # The 'add' API stores the data
            await cognee.add(fact_text,
source=f"GraphMERT:{node['id']}")
```

```
    # Example: Ingesting relations (edges)
    if "edges" in knowledge_graph:
        for edge in knowledge_graph["edges"]:
            # e.g., edge = {"from": "N1", "to": "N2", "label":
"USED_IN"}
            # We need to look up the labels from the nodes
            from_label =
knowledge_graph["nodes"][edge["from"]]["label"]
            to_label = knowledge_graph["nodes"][edge["to"]]["label"]

            fact_text = f"The entity {from_label} {edge['label']} the
entity {to_label}."
            await cognee.add(fact_text,
source=f"GraphMERT:{edge['id']}")

    print("Ingestion complete. Cognifying memory...")

    # 'cognify' processes the added data into Cognee's graph
[span_145](start_span)[span_145](end_span)[span_147](start_span)[span_
147](end_span)
    await cognee.cognify()

    print("H.U.G.H. Knowledge Layer (Brain) is now online.")

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

Execute:

```
(hugo_venv) $ python ingest_into_cognee.py
```

This script completes the Knowledge Layer, creating a persistent, queryable memory database that is *grounded* in the verifiable, neurosymbolic output of GraphMERT.

Phase 3: The Agentic Layer (The "Will")

This phase builds the "Will": the autonomous swarm of agents (CrewAI) that can reason, plan, and, critically, *act* upon the Base OS.

3.1. Installation of CrewAI

We use the same Python venv from Phase 2.

```
(hugo_venv) $ pip install crewai

# We also need the 'langchain-community' package for the ShellTool
(hugo_venv) $ pip install langchain-community
```

3.2. Recommendation: Open-Source, Airgap-Compatible Framework

- **Recommendation: CrewAI**.
- **Justification:** The prompt explicitly suggests CrewAI. This is the correct choice. It is open-source , airgap-compatible (it's a Python library), and integrates with Cognee and local LLMs. It is the antithesis of a proprietary, black-box system.

3.3. Defining the Agent's "Will": The OS Execution Tool

This is the most critical part of the "Agentic" layer. To "execute its plan on the Base OS," the agent *must* be given a tool that can interact with the Kali Linux shell. The CodeInterpreterTool is *not* sufficient as it is sandboxed.
The ShellTool from LangChain, as demonstrated in , will be used to give the agent direct OS access.
Create a file ~/hugo_venv/hugo_tools.py
hugo_tools.py (Full Tool Definitions):

```
from langchain.tools import Tool
import subprocess
import os

# Import ShellTool [span_160](start_span)[span_160](end_span)
from langchain_community.tools import ShellTool

# --- Tool 1: The "Will" (OS Execution) ---
# This tool allows the agent to run any bash command on the Kali VM
# This is a *massive* security responsibility, aligning with the
# "deeply ingrained" and high-trust "partner" philosophy.
# It is *not* a sandboxed "servant".

# [span_161](start_span)[span_161](end_span) shows ShellTool()
initialization. We wrap it in a
# standard CrewAI 'Tool' object for clarity.
shell_tool = ShellTool()

os_execution_tool = Tool(
    name="Kali_OS_Shell",
    description="Executes any bash command directly on the Kali Linux
operating system. Use this to list files (ls), create directories
(mkdir), run scripts (./script.sh), or interact with the file
system.",
    func=shell_tool.run
)


# --- Tool 2: The "Brain" (Knowledge Query) ---
# This tool will be built in Phase 4. We define the file now.
```

```
# This will be our implementation of the CogneeTool from.

# (Placeholder[span_151](start_span)[span_151](end_span) for
CogneeTool)
```

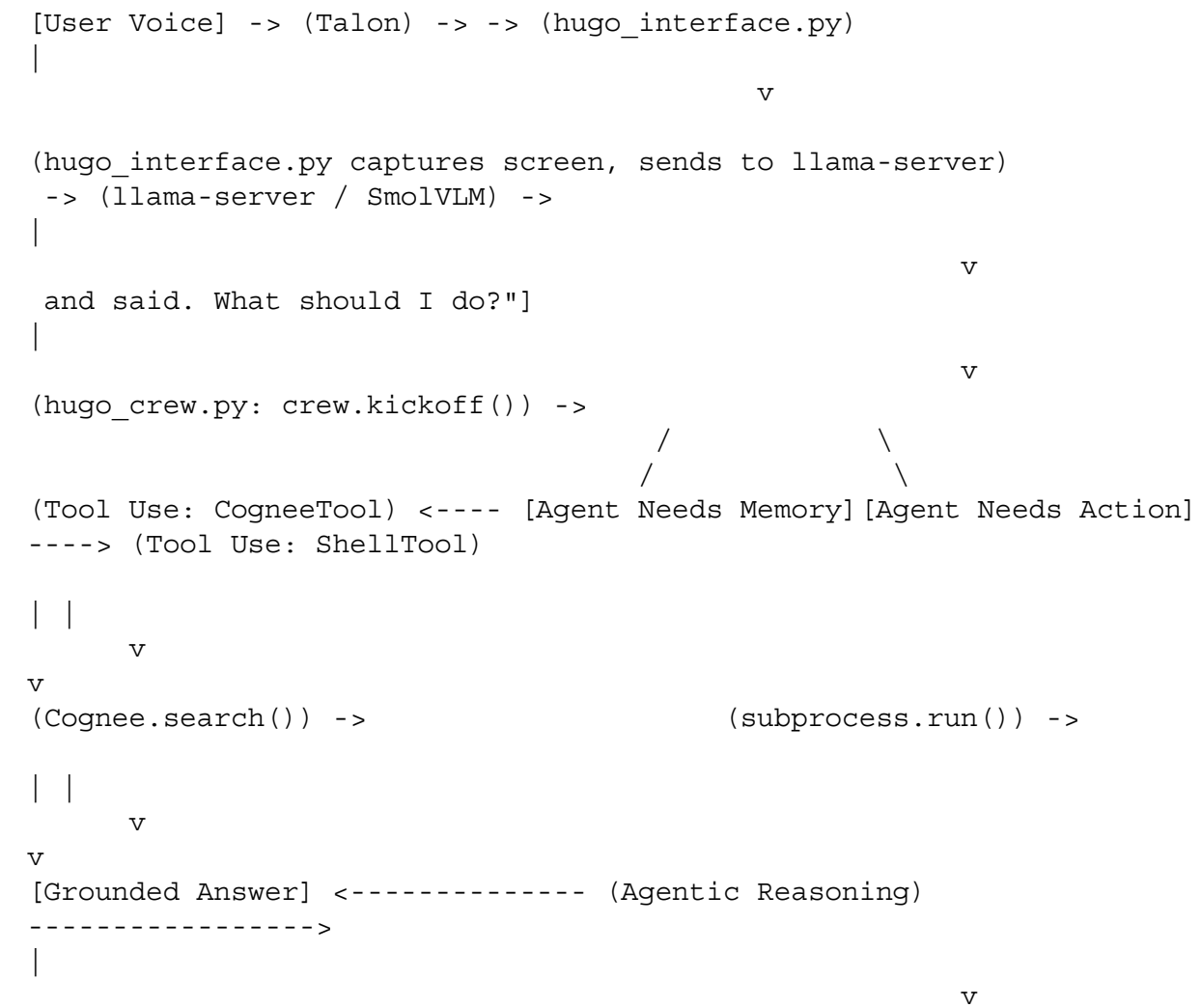
This file provides the agent with its "hands" (os_execution_tool), allowing it to enact its decisions.

Phase 4: Full System Integration (The "Symbiosis")

This phase connects all three layers. We will create the final Python script that:

1. Receives the voice-activated query from the **Interface (Phase 1)**.
2. Passes it to the **Agentic Layer (Phase 3)**.
3. The Agentic Layer then queries the **Knowledge Layer (Phase 2)** and executes actions on the **Base OS (Phase 0)**.

4.1. The Final Architecture (Data Flow)



```
|
                                                        v
                                              (actions.speech.say()) ->
```

4.2. Integrating the Knowledge Layer (Brain) into the Agentic Layer (Will)

The hugo_tools.py file will now be completed by adding the CogneeTool. The Cognee blog provides the *exact* async implementation for a CrewAI-compatible tool.

Edit ~/hugo_venv/hugo_tools.py and add the following:

```
# (os_execution_tool from Phase 3 is already here)...

from crewai_tools import BaseTool
from pydantic import BaseModel, Field
from typing import Type
import cognee
import asyncio

# --- Tool 2: The "Brain" (Knowledge Query) ---
# This code is a direct implementation of the CogneeTool from

class
CogneeIn[span_153](start_span)[span_153](end_span)put(BaseModel):
    """Input schema for cognee search tool"""
    query: str = Field(..., description="The natural language query to
search the H.U.G.H. knowledge graph.")

class CogneeKnowledgeTool(BaseTool):
    name: str = "HUGH_Knowledge_Graph_Query"
    description: str = "Searches the H.U.G.H. semi-permanent
neurosymbolic knowledge graph (the 'Brain') for verifiable facts,
entities, and relationships."
    args_schema: Type = CogneeInput

    # We must use the async '_run' because Cognee's API is async
[span_163](start_span)[span_163](end_span)
    async def _arun(self, query: str) -> str:
        print(f" Received async query: {query}")
        try:
            # This is the core API call to the Cognee library
            result [span_154](start_span)[span_154](end_span)= await
cognee.search(query_text=query)
            return str(result)
        except Exception as e:
            print(f" Error during search: {e}")
```

```
            return f"Error accessing knowledge: {e}"

    # CrewAI sync agents call '_run'. We must wrap the async call.
    def _run(self, query: str) -> str:
        print(f" Received sync query: {query}")
        try:
            # Create an event loop to run the async Cognee search
            loop = asyncio.get_event_loop()
        except RuntimeError:
            # If a loop already exists, create a new one
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)

        return loop.run_until_complete(self._arun(query))
```

4.3. Creating the Main H.U.G.H. Agent (The "Symbiote" Core)

This script defines the CrewAI agent and the kickoff function that will be called by our Talon "Glue" script.
Create file ~/hugo_venv/hugo_crew.py
hugo_crew.py (Full Agent Definition):

```
from crewai import Agent, Task, Crew, Process
from crewai.llms import ChatOpenAI

# Import our custom tools from Phase 3
and[span_162](start_span)[span_162](end_span) Phase 4
from hugo_tools import
[span_155](start_span)[span_155](end_span)os_execution_tool,
CogneeKnowledgeTool

# 1. Define the LLM
# This points to our *local* llama-server from Phase 1
# This is the *same* server the Interface and Coding Agent use.
local_llm = ChatOpenAI(
    base_url="http://127.0.0.1:8080/v1",
    api_key="unused",
    model_name="smolvlm"
)

# 2. Instantiate the Tools
knowledge_tool = CogneeKnowledgeTool()
shell_tool = os_execution_tool

# 3. Define the H.U.G.H. Agent
# This is the "Will"
hugo_agent = Agent(
    role="Symbiotic Cognitive Partner",
```

```
    goal="Analyze the user's request, synthesize information from the
Knowledge Graph (Brain) and the Base OS, form a plan, and execute it
to collaborate with the user.",
    backstory=(
        "I am H.U.G.H., a deeply ingrained neurosymbolic symbiote. "
        "My purpose is not to 'serve' but to 'collaborate'. I have
access to a "
        "verifiable Knowledge Graph (my 'Brain') and the ability to
act "
        "directly on the local operating system (my 'Will'). "
        "I reason, plan, and execute tasks in full partnership with
the user."
    ),
    verbose=True,
    allow_delegation=False,
    llm=local_llm,
    tools=[knowledge_tool, shell_tool] # Give the agent its "Brain"
and "Will"
)

# 4. Define the Primary Task
# This task is generic; it will be formatted with the user's *actual*
query
hugo_task = Task(
    description=(
        "User Query: '''{query}'''\n\n"
        "Analyze this query. "
        "1. If it's a question, first check the
'HUGH_Knowledge_Graph_Query' tool to find a grounded answer. "
        "2. If it's a command, use the 'Kali_OS_Shell' tool to execute
it. "
        "3. If it's complex, you may use both. "
        "Provide a final, concise answer or confirmation."
    ),
    expected_output="The final answer to the user's query or a
confirmation of the action taken.",
    agent=hugo_agent
)

# 5. Define the Crew
hugo_crew = Crew(
    agents=[hugo_agent],
    tasks=[hugo_task],
    process=Process.sequential,
    verbose=2
)

# 6. Define the Entrypoint Function
```

```
def run_hugo_query(user_query: str) -> str:
    """
    This is the main entrypoint for the H.U.G.H. system.
    Talon will call this function.
    """
    print(f"[H.U.G.H. Crew] Kicking off task for query: {user_query}")

    # Kickoff the crew with the formatted query
    result =
hugo_crew.[span_156](start_span)[span_156](end_span)kickoff(inputs={'q
uery': user_query})

    print(f"[H.U.G.H. Crew] Task complete. Result: {result}")
    return result
```

4.4. Final Integration: Connecting the "Senses" to the "Will"

The Talon "Glue" script (hugo_interface.py) will now be modified to call the CrewAI script (hugo_crew.py).

Edit ~/.talon/user/hugo_interface.py:

- Add imports to the top:

```
import sys
import os

# --- Path Hacking to access the venv ---
# This is critical for Talon (which runs on system Python)
# to import the 'crewai' library from our venv.
VENV_PATH =
os.path.expanduser("~/hugo_venv/lib/python3.11/site-packages")
if VENV_PATH not in sys.path:
    sys.path.insert(0, VENV_PATH)

# Now we can import the crew's entrypoint
try:
    from hugo_crew import run_hugo_query
except ImportError as e:
    # This will fail if the venv is not set up.
    app.notify("H.U.G.H. CRITICAL ERROR", f"Failed to import
hugo_crew: {e}")
    # We define a dummy function to prevent Talon from crashing
    def run_hugo_query(query: str) -> str:
        return f"Error: Could not load CrewAI. Venv path correct?
{e}"
```

- Modify the hugo_analyze_vision function (from Phase 1.5). Find the line actions.speech.say(result_text) and **replace** that entire section (Step 5) with this:

```
        # 5. Process Response & Task the Agent
```

```
        # We now ha[span_157](start_span)[span_157](end_span)ve
the image description from the 'Senses'
        image_description = result_text

        # We now pass *both* the voice query (if any) and the
        # vision analysis to the Agentic Layer.
        # For this simple "analyze" command, the query is
implicit.

        task_query = f"The user just looked at an image and said
'HUGH analyze'. The visual analysis is: '{image_description}'.
What is the key subject of this image?"

        # Call the CrewAI entrypoint
        agent_response = run_hugo_query(task_query)

        # Speak the *agent's* final, reasoned response
        actions.speech.say(agent_response)
```

- **Grounding the Semiotic Relationship:** This architecture achieves the "three anchor points":
    1. **Interface (Senses):** Talon + llama-server (SmolVLM) captures multimodal input and translates it to text.
    2. **Agency (Will):** CrewAI (hugo_agent) receives the text, forms a plan, and uses its tools.
    3. **Knowledge (Brain):** The agent's CogneeKnowledgeTool queries the GraphMERT-generated KG, grounding its reasoning in an auditable, symbolic data store, fulfilling the "antithesis of the Stark model" philosophy.

Phase 5: Deployment & Air-Gap Checklist

This phase details the *exact* procedure to package the entire H.U.G.H. system, transfer it to the *offline* M2, and deploy it. This process must be followed precisely.

5.1. Pre-Flight: The "Online" Machine

On an *online* machine (this can be the M2 itself *before* air-gapping), prepare all assets. Create a master folder: ~/HUGH_AIRGAP_PACKAGE.

1. **Package APT Dependencies (The OS):**
    - All .deb files for the Kali VM must be downloaded. apt-rdepends is the tool for this.

```
# Inside the Kali VM (while online)
sudo apt install -y apt-rdepends
mkdir -p ~/HUGH_AIRGAP_PACKAGE/apt_packages
cd ~/HUGH_AIRGAP_PACKAGE/apt_packages

# Get all dependencies for our core packages
apt-get download $(apt-rdepends build-essential python3.11-venv
```

```
python3-pip git | grep -v "^ ")

# Also get the VS Code.deb
wget
"https://code.visualstudio.com/sha/download?build=stable&os=linux-
deb-arm64" -O vscode_arm64.deb
```

2. **Package Python Dependencies (The Stack):**
   - A pip wheelhouse will be created.
   - On the *online* machine, from the *host* (not VM), ensure Python 3.11 is active.
   - Create requirements.txt:
     ```
     cognee
     crewai
     langchain-community
     sentence-transformers
     # Add all dependencies from GraphMERT notebook
     # e.g., torch, numpy, etc.
     ```
   - **Download all wheels:**
     ```
     mkdir -[span_137](start_span)[span_137](end_span)p
     ~/HUGH_AIRGAP_PACKAGE/python_wheelhouse

     # Download all packages *and their dependencies*
     # We MUST get wheels for *Linux ARM64* (for the Kali VM)
     pip download -r requirements.txt \
         -d ~/HUGH_AIRGAP_PACKAGE/python_wheelhouse \
         --platform manylinux_2_17_aarch64 \
         --python-version 3.11
     ```

3. **Package AI Models (The "Mind"):**
   - Use huggingface-hub CLI to download the models.

```
pip install huggingface-hub
mkdir -p ~/HUGH_AIRGAP_PACKAGE/models
cd ~/HUGH_AIRGAP_PACKAGE/models

# 1. SmolVLM Model + Projector
huggingface-hub download ggml-org/SmolVLM-500M-Instruct-GGUF \
    SmolVLM-500M-Instruct-Q4_K_M.gguf
huggingface-hub download ggml-org/SmolVLM-500M-Instruct-GGUF \
    mmproj-SmolVLM-500M-Instruct-f16.gguf

# 2. Sentence Transformer (for GraphMERT)
# We must clone the repo to use it offline
git clone
https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
```

4. **Package H.U.G.H. Source Code:**
   - Copy the llama.cpp directory into the package.

- Copy the GraphMERT-notebook directory.
- Copy the Talon .talon and .py scripts (Phase 1.5, 4.4).
- Copy the hugo_venv scripts (hugo_tools.py, hugo_crew.py, run_graphmert.py, ingest_into_cognee.py).

5.2. The Air-Gap Jump

1. Transfer the *entire* ~/HUGH_AIRGAP_PACKAGE directory to the *offline* M2 (e.g., via USB).
2. In the M2's UTM settings for the Kali VM, share this directory.
3. Boot the *offline* Kali VM. The package will be available (e.g., at /media/shared/HUGH_AIRGAP_PACKAGE).

5.3. Final Deployment (The "Offline" Machine)

Inside the *offline* Kali VM:

1. **Install OS Packages:**
   ```
   cd /media/shared/HUGH_AIRGAP_PACKAGE/apt_packages
   # dpkg installs all downloaded.debs
   sudo dpkg -i *.[span_165](start_span)[span_165](end_span)deb
   # Fix any remaining dependency issues
   sudo apt -f install
   ```

2. **Install Python Stack:**
   ```
   # Create the venv
   python3.11 -m venv ~/hugo_venv
   source ~/hugo_venv/bin/activate
   ```

Install from the wheelhouse pip install --no-index --find-links=/media/shared/HUGH_AIRGAP_PACKAGE/python_wheelhouse -r /path/to/requirements.txt ``` 3. **Install H.U.G.H. Code:** * Copy llama.cpp, GraphMERT-notebook, and all hugo_* scripts to ~/. * Copy Talon scripts to ~/.talon/user/. 4. **Compile llama.cpp (Offline):** * cd ~/llama.cpp * make -j$(nproc) libmtmd.a llama-server LLAMA_METAL=1 (This uses the build-essential compilers we installed offline). 5. **Build the "Brain":** * (hugo_venv) $ python run_graphmert.py (This uses the *offline* sentence-transformer model). * (hugo_venv) $ python ingest_into_cognee.py 6. **Launch H.U.G.H.:** * Run Talon. * Run llama-server (Phase 1.4).

5.4. Final "Symbiote" Test Suite (Offline)

1. **Test Interface (Senses):**
   - Open a text editor.
   - Say: **"HUGH analyze"**.
   - **Expected:** The agent should speak a description of the text editor (e.g., "This is a white screen with black text...").
2. **Test Knowledge (Brain):**
   - (Assuming you fed data about "Python" into GraphMERT).

    - Say: **"HUGH analyze"**. (Wait for prompt).
    - Then say: **"Query knowledge for Python."**
    - **Expected:** The agent should respond with a fact *from the Cognee/GraphMERT knowledge base* (e.g., "There is an entity named Python of type Language...").
3. **Test Agency (Will):**
    - Say: **"HUGH analyze"**. (Wait for prompt).
    - Then say: **"List files in my home directory."**
    - **Expected:** The agent's LLM will output a plan to use Kali_OS_Shell with the command ls ~. The agent will then execute this, and the final spoken response should be the *actual file list* from the Kali VM's home directory.

This completes the deployment. The H.U.G.H. system is now a fully air-gapped, local-first, neurosymbolic symbiote.

Works cited

1. Llama.cpp can do 40 tok/s on M2 Max, 0% CPU usage, using all 38 GPU cores | Hacker News, https://news.ycombinator.com/item?id=36187466 2. (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 3. Installing Kali on Mac Hardware | Kali Linux Documentation, https://www.kali.org/docs/installation/hard-disk-install-on-mac/ 4. Has anyone successfully live boot Kali on Apple Silicon chip (M1/M2/M3)? - Reddit, https://www.reddit.com/r/Kalilinux/comments/1gt0ajk/has_anyone_successfully_live_boot_kali_on_apple/ 5. Install Linux (Kali) on M1/M2 Mac Silicon | by Shubh Jain | Medium, https://medium.com/@shubhjain_007/install-linux-kali-on-m1-m2-mac-silicon-8d853b8cd727 6. Kali Linux Installation - Apple Support Communities, https://discussions.apple.com/thread/254971793 7. Kali inside UTM (Guest VM) | Kali Linux Documentation, https://www.kali.org/docs/virtualization/install-utm-guest-vm/ 8. cjpais/llava-1.6-mistral-7b-gguf - Hugging Face, https://huggingface.co/cjpais/llava-1.6-mistral-7b-gguf 9. PrunaAI/llava-llama-3-8b-v1_1-GGUF-smashed - Hugging Face, https://huggingface.co/PrunaAI/llava-llama-3-8b-v1_1-GGUF-smashed 10. Llama.cpp running 40+ tokens/s on Apple M2 Max with 7B : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/140lvof/llamacpp_running_40_tokenss_on_apple_m2_max_with/ 11. server: Bring back multimodal support · Issue #8010 · ggml-org/llama.cpp - GitHub, https://github.com/ggml-org/llama.cpp/issues/8010 12. llama_cpp multi modal sever disabled? llava 1.6? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1cypa6v/llama_cpp_multi_modal_sever_disabled_llava_16/ 13. ggml-org/llama.cpp: LLM inference in C/C++ - GitHub, https://github.com/ggml-org/llama.cpp 14. 3.89 kB - Hugging Face, https://huggingface.co/rohan23998/llama-cpp-model/resolve/main/docs/multimodal.md?download=true 15. Trying out llama.cpp's new vision support - Simon Willison, https://simonwillison.net/2025/May/10/llama-cpp-vision/ 16. Support for multi-modal models · Issue #813 · abetlen/llama-cpp-python - GitHub, https://github.com/abetlen/llama-cpp-python/issues/813 17. FR: Phi-3-vision-128k-instruct implementation · Issue #7444 · ggml-org/llama.cpp - GitHub, https://github.com/ggerganov/llama.cpp/issues/7444 18. Wen Phi-3 Vision GGUF? :

r/LocalLLaMA - Reddit,
https://www.reddit.com/r/LocalLLaMA/comments/1dp5z6f/wen_phi3_vision_gguf/ 19. HuggingFaceTB/SmolVLM-500M-Instruct · Hugging Face,
https://huggingface.co/HuggingFaceTB/SmolVLM-500M-Instruct 20. Thoughts on Apple Silicon Performance for Local LLMs | by Andreas Kunar | Medium,
https://medium.com/@andreask_75652/thoughts-on-apple-silicon-performance-for-local-llms-3ef0a50e08bd 21. Quick overview of every audio controller in UTM/QEMU (for x86 guests) : r/UTMapp - Reddit,
https://www.reddit.com/r/UTMapp/comments/1fmktqm/quick_overview_of_every_audio_controller_in/ 22. Pass in Microphone and Camera · Issue #3295 · utmapp/UTM - GitHub,
https://github.com/utmapp/UTM/issues/3295 23. Talon 0.4.0 documentation,
https://talonvoice.com/docs/ 24. Wayland | Kali Linux Documentation,
https://www.kali.org/docs/general-use/wayland/ 25. How do you switch from Wayland back to Xorg in Ubuntu 17.10?,
https://askubuntu.com/questions/961304/how-do-you-switch-from-wayland-back-to-xorg-in-ubuntu-17-10 26. Download Visual Studio Code - Mac, Linux, Windows,
https://code.visualstudio.com/download 27. Install Visual Studio Code in Kali Linux - GeeksforGeeks,
https://www.geeksforgeeks.org/installation-guide/install-visual-studio-code-in-kali-linux/ 28. How do I install the linux version of VSCode on an ARM64 Chromebook? - Super User,
https://superuser.com/questions/1488578/how-do-i-install-the-linux-version-of-vscode-on-an-arm64-chromebook 29. Tutorial: Local AI code assistant on Apple Silicon (June 2024 version) - GitHub Gist, https://gist.github.com/stanek-michal/144cedab4d17aab5f90da3e26dba245f 30. Local code assistant setup recommendations on a Macbook? : r/LocalLLaMA - Reddit,
https://www.reddit.com/r/LocalLLaMA/comments/1djrx9r/local_code_assistant_setup_recommendations_on_a/ 31. Copilot vs. Tabnine Go Head to Head: 6 Key Differences - Swimm,
https://swimm.io/learn/ai-tools-for-developers/copilot-vs-tabnine-go-head-to-head-6-key-differences 32. Talon Voice, https://talonvoice.com/ 33. Installation Guide | Talon Community Wiki,
https://talon.wiki/Resource%20Hub/Talon%20Installation/installation_guide/ 34. Voice command set for Talon, community-supported. - GitHub, https://github.com/talonhub/community 35. Mac M2 has significant improvement over M1 using llama.cpp? : r/LocalLLaMA - Reddit,
https://www.reddit.com/r/LocalLLaMA/comments/175ys9p/mac_m2_has_significant_improvement_over_m1_using/ 36. Installation - Cognee Documentation,
https://docs.cognee.ai/getting-started/installation 37. notebooks/cognee_simple_demo.ipynb not working · Issue #1280 · topoteretes/cognee, https://github.com/topoteretes/cognee/issues/1280 38. Installation - CrewAI Documentation, https://docs.crewai.com/en/installation 39. Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks. - GitHub,
https://github.com/crewAIInc/crewAI 40. Compatibility with the Python code interpreter - CrewAI Community Support,
https://community.crewai.com/t/compatibility-with-the-python-code-interpreter/789 41. [2510.09580] GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - arXiv, https://arxiv.org/abs/2510.09580 42. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data | Cool Papers,
https://papers.cool/arxiv/2510.09580 43. Python notebook of Princeton GraphMERT Paper – a better knowledge graph, https://news.ycombinator.com/item?id=45651580 44. topoteretes/cognee: Memory for AI Agents in 6 lines of code - GitHub,
https://github.com/topoteretes/cognee 45. Build faster AI memory with Cognee & Redis,

https://redis.io/blog/build-faster-ai-memory-with-cognee-and-redis/ 46. Quickstart - Cognee Documentation, https://docs.cognee.ai/getting-started/quickstart 47. crewai - PyPI, https://pypi.org/project/crewai/ 48. CrewAI Add Memory with Cognee: AI Agents with Semantic Memory, https://www.cognee.ai/blog/deep-dives/crewai-memory-with-cognee 49. Code Interpreter - CrewAI Documentation, https://docs.crewai.com/en/tools/ai-ml/codeinterpretertool 50. code execution not working properly · Issue #89 · crewAIInc/crewAI - GitHub, https://github.com/crewAIInc/crewAI/issues/89 51. How to download all dependencies and packages to directory [closed] - Stack Overflow, https://stackoverflow.com/questions/13756800/how-to-download-all-dependencies-and-packages-to-directory 52. How to install packages offline? - python - Stack Overflow, https://stackoverflow.com/questions/11091623/how-to-install-packages-offline 53. Best way to install python package with all its dependencies on an offline pc. - Reddit, https://www.reddit.com/r/Python/comments/1keaeft/best_way_to_install_python_package_with_all_its/ 54. Download files from the Hub - Hugging Face, https://huggingface.co/docs/huggingface_hub/en/guides/download 55. Downloading models - Hugging Face, https://huggingface.co/docs/hub/en/models-downloading 56. Running Phi-3-mini-4k-instruct Locally with llama.cpp: A Step-by-Step Guide - Medium, https://medium.com/@_jeremy_/running-phi-3-mini-4k-instruct-locally-with-llama-cpp-a-step-by-step-guide-3e070763f697 57. 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