H.U.G.H. Implementation Blueprint: A Phased Technical Guide

Architect's Foreword: The Philosophy and Primary Constraints

This document provides a comprehensive, phased implementation plan for Project H.U.G.H. (Holo-Utopic Guardian Heuristic). It is designed to guide a proficient developer from a fresh Kali Linux installation to a fully deployed, air-gapped agentic operating system.

On Philosophy: This blueprint adheres strictly to the specified "Philosophical Northstar". The architecture is designed to be auditable, personal, and symbiotic. We reject the "black-box" model of proprietary systems like J.A.R.V.I.S.. Every component selected is open-source, and every data flow is explicit. The goal is a "partner," not a "product." This philosophy informs our technical choices: compiling from source for transparency, prioritizing local inference for privacy, and building a verifiable knowledge graph for trust.

The Primary Constraint: 8GB Unified RAM: Before proceeding, all implementation steps must be understood through the lens of our most significant hard constraint: the 8GB M2 Unified RAM. This is not merely a "low-spec" machine; it is a high-performance system with an *extremely* tight memory budget.

The M2's Unified RAM is both an advantage and a liability. It is high-bandwidth and shared directly between the CPU and GPU. This eliminates the data-copy latency associated with discrete VRAM but also means our operating system, Python scripts, agent framework, and Large Language Model (LLM) are all competing for the same 8GB pool.

A standard Kali Linux GNOME desktop will consume approximately 1.5-2GB of RAM on boot. The various Python processes for Talon, Cognee, and CrewAI will add several hundred megabytes. This leaves a functional budget of approximately 5-6GB for the core multimodal model. A 4-bit quantized (GGUF) 3B-to-4B parameter model, such as the selected Phi-3-Vision , will require approximately 2.5-4GB of RAM to load. This leaves a razor-thin margin of 1-2GB for context, the KV cache, and all other system operations.

This entire project is therefore feasible *only* with aggressive 4-bit quantization (e.g., Q4_K_M GGUF) and 100% GPU offload to the M2's Metal cores. Any deviation, such as attempting to use a 7B model or failing to compile with full Metal support, will result in immediate system failure via memory thrashing. This constraint dictates our technical choices in every phase.

Critical Deviation Note: The GraphMERT Problem: The prompt specifies the 80M-parameter GraphMERT model as the neurosymbolic core. Analysis of the corresponding research confirms this is a *closed-source Princeton research project*. The model weights are not publicly available, and there is no open-source repository for implementation.

As the architect, a viable, open-source, and lightweight replacement pipeline has been sourced to achieve the same *goal* (text-to-KG extraction) while respecting our constraints. Using our primary multimodal model (Phi-3-Vision) for this task via a library like LlamaIndex is possible , but this would be a heavy, slow process that would overload our single LLM.

A superior architecture involves specialized models. This blueprint will therefore substitute the "GraphMERT" task with a more resilient hybrid pipeline:

1. **COMET-BART:** A small, fast, pre-trained transformer from the Allen Institute , designed *specifically* for generating commonsense knowledge graphs. The kogito Python library provides a direct, simple interface for this: text -> kogito(COMET-BART) -> kgraph.
2. **LlamaIndex KnowledgeGraphIndex:** This will be used for deeper, *domain-specific* entity extraction during non-real-time data ingestion , configured to use a local, air-gapped LLM.
3. **Cognee:** Both pipelines will feed their extracted triples into the Cognee memory library , which remains the "Brain's" primary interface as requested. This achieves the neurosymbolic goal within our open-source and hardware constraints.

Phase 0: Environment Setup & System Tooling

This phase lays the foundational bedrock of the system. A single misstep here—particularly regarding the display server or Python environment—will cause cascading failures in later phases.

0.1 OS Installation: Kali Linux on Apple Silicon (Asahi)

The developer will begin on a standard macOS installation. The Asahi Linux project is the upstream for all Apple Silicon Linux support and provides the installer.

1. From the macOS terminal, execute the official Asahi installer, which handles all partitioning:
   ```
   curl https://alx.sh | sh
   ```
2. During the installation process, the developer will be prompted to choose a distribution. They must select the **Kali Linux (ARM64)** image. This will correctly install the Asahi kernel and drivers with the Kali userland.
3. The default desktop environment (DE) for the Kali Asahi image is typically GNOME or KDE. This plan will assume GNOME, but the X11 requirement detailed below is universal.

0.2 Critical Sub-Task: Forcing X11 for Talon Compatibility

This is the first and most critical, non-obvious failure point in the build.

- **Problem:** The user *requires* Talon for voice control. Talon's Linux support *explicitly requires an X11 session* to function. Its screen-reading and input-hooking APIs are not compatible with the Wayland display server.
- **Conflict:** Modern Linux distributions, including Asahi/Kali, default to Wayland for its improved performance and security.
- **Resolution:** Attempting to install Talon in the default Wayland session will result in immediate failure. The developer must force an X11 session *before* proceeding.

Implementation Steps:

1. **Verify Current Session:** After installing Kali and booting to the desktop, open a terminal and verify the session type.
   ```
   echo $XDG_SESSION_TYPE
   ```
   If this command returns wayland , the developer must switch. If it returns x11, no action is needed.
2. **Switch to X11 Session:** The simplest, non-destructive method is from the login screen (GDM or SDDM).

    - Instruct the developer to **log out**.
    - On the login screen, *before* entering the password, locate the **"cogwheel" icon** (typically in the bottom-right corner).
    - Click this icon and select the session named **"GNOME on Xorg"** or **"Plasma (X11)"**.
    - Log in. This session will now be X11-based, and the system will remember this preference for future logins.
3. **Troubleshooting (If X11 is Missing):** If the X11 session option is not available, the core components may be missing. Install the full metapackage to ensure all dependencies, including Xorg, are present.
   ```
   sudo apt update
   sudo apt install -y kali-desktop-gnome
   ```
   This kali-desktop-gnome metapackage includes all necessary components for the GNOME DE.

0.3 System & Python Tooling (The ARM64 "build-essentials")

The Python environment must be handled with care. On a specialized distribution like Kali, using sudo pip install is a fast path to breaking critical system dependencies. Kali 2024.4 and later actively block this practice with an externally-managed-environment error.
Therefore, all Python tools will be installed using pipx (for global tools) and venv (for project-specific dependencies).
Implementation Steps:

```
# Update and install core build tools for the ARM64 architecture
sudo apt update
sudo apt install -y build-essential git cmake libopenblas-dev

# Install Python 3, pip, and the safe-environment tools
sudo apt install -y python3-pip python3-venv python3-pipx

# Configure pipx (run as user, not root)
pipx ensurepath
```

The core packages required for the H.U.G.H. build are detailed in the table below.

Package	Purpose	Install Command
build-essential	C/C++ compilers, make, etc. Critical for compiling llama.cpp.	sudo apt install -y build-essential
cmake	Build system required by llama.cpp.	sudo apt install -y cmake
git	For cloning llama.cpp and other project repositories.	sudo apt install -y git
python3-pip	The Python package installer.	sudo apt install -y python3-pip
python3-venv	For creating isolated Python environments, as mandated by PEP 668.	sudo apt install -y [span_48](start_span)[span_48](end_span)python3-venv
python3-pipx	For safely installing Python-based system tools (like	sudo apt install -y [span_49](start_span)[span_49]

Package	Purpose	Install Command
	the CrewAI CLI).	(end_span)python3-pipx
libopenblas-dev	Provides optimized numerical operations (BLAS) required by llama.cpp for CPU fallback.	sudo apt install -y libopenblas-dev

0.4 VS Code & Local-First Coding Agent

VS Code Installation:

1. Navigate to the official Visual Studio Code website and download the **ARM64 .deb package**.
2. Install the package using apt, which correctly handles all dependencies:
   ```
   # Navigate to the Downloads folder
   cd ~/Downloads
   sudo apt install./code_*.arm64.deb
   ```

Critical Recommendation: Coding Agent (Tabby)
The H.U.G.H. philosophy demands an open-source, local-first stack. **Tabby** is the superior choice for this project. It is self-hosted, provides a VS Code extension, and—most importantly—has explicit **Apple M1/M2 Metal support**. This makes it maximally efficient on the target hardware, aligning perfectly with the 8GB RAM constraint and the final air-gap requirement.
Installation (via Docker, pre-air-gap):

1. (This step assumes Docker is installed on the Asahi/Kali build.)
2. Pull and run the Tabby server, specifying the metal device for GPU acceleration.
   ```
   docker run -it -p 8080:8080 --gpus all \
     tabbyml/tabby \
     serve --model TabbyML/StarCoder-1B --device metal
   ```

3. In VS Code, install the "Tabby" extension from the marketplace and configure it to point to the local server at http://localhost:8080.

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

This layer is the primary I/O for the symbiotic relationship. It must be low-latency and fully local.

1.1 Talon Voice Installation & Configuration

1. The developer will download the **Talon for Linux** .tar.xz file from the official Talon website.
2. Extract the archive. Talon is self-contained and runs from its extracted directory. The configuration lives in ~/.talon.
   ```
   tar -xf Talon-linux-*.tar.xz
   cd talon
   ```

./talon # First run 3. On first run, a tray icon will appear. The developer must click this icon and select **"Speech Recognition" -> "Conformer"**.[span_62](start_span)[span_62](end_span) This will download and install Talon's free, high-performance, local speech-to-text engine. 4.

Install the Talon Community scripts, which provide the basic command grammar (e.g., "talon sleep," "talon wake"):bash cd ~/.talon/user git clone https://github.com/talonhub/knausj_talon.git ``` 5. Restart Talon. The developer can now test the voice engine by saying "talon sleep."

1.2 Inference Engine: Compiling llama.cpp with Apple Metal

This is the core of the system's performance strategy. We *cannot* use a pre-built binary or a simple pip install. We *must* compile llama.cpp from source to enable the Apple Metal (GPU) backend. This step is non-negotiable for meeting the 8GB RAM constraint.

Implementation Steps:

1. Clone the llama.cpp repository:
   ```
   git clone https://github.com/ggml-org/llama.cpp.git
   cd llama.cpp
   ```

2. Create a dedicated Python virtual environment for the inference engine:
   ```
   python3 -m venv venv
   source venv/bin/activate
   pip install -r requirements.txt
   ```

3. Compile llama-cpp-python with Metal support. The CMAKE_ARGS environment variable passes the necessary build flags to the compiler.
   ```
   # This is the single most important command for system performance
   CMAKE_ARGS="-DLLAMA_METAL=on" FORCE_CMAKE=1 pip install -e.
   ```
   This command builds the Python bindings with full Metal acceleration, allowing the M2's GPU to handle all model inference.

1.3 Multimodal Model: Phi-3-Vision (GGUF)

The model choice is dictated by the 8GB RAM constraint. We select **Phi-3-Vision-128k-Instruct** over alternatives like LLaVA.

- **Performance:** Phi-3 is a 4.2B parameter model. Benchmarks for similar small models (2B) show 15-20 tokens/second on 8GB M1 systems. We can confidently project a usable, real-time performance of 5-15 t/s from a 4-bit quantized Phi-3 on the M2's Metal backend.
- **Feasibility:** A 4-bit quant (Q4_K_M) will consume approximately 3-4GB of RAM. This leaves just enough headroom for the OS and agentic layers. A 7B LLaVA model would require >4.5GB and would likely cause the system to thrash and become unusable.

Table: Multimodal Model 8GB M2 Performance Analysis

Model	Quantization	Size (Params)	Est. RAM Usage	Est. Tokens/Sec (Metal)	Feasibility
LLaVA v1.5	Q4_K_M	7B	~4.5-5.5 GB	< 5 t/s	Very Low (High risk of thrashing)
Phi-3-Vision	**Q4_K_M**	4.2B	**~3-4 GB**	**~5-15 t/s**	**High (Optimal Choice)**

The GGUF model file will be pre-downloaded as part of the Phase 5 air-gap procedure.

1.4 The "Senses" Server: Launching llama.cpp

The developer will now launch the local inference server, which provides an OpenAI-compatible API. This is the endpoint that all other H.U.G.H. components will communicate with.
Server-Launch Command:

```
# (Activate the llama.cpp venv first)
source ~/llama.cpp/venv/bin/activate

# --n_gpu_layers -1 = OFFLOAD ALL LAYERS TO METAL. This is critical.
# --n_ctx 4096 = Set context size.
# --host 127.0.0.1 = Bind to localhost only.
python3 -m llama_cpp.server \
  --model /path/to/models/phi-3-vision-128k-instruct.Q4_K_M.gguf \
  --n_gpu_layers -1 \
  --n_ctx 4096 \
  --host 127.0.0.1 \
  --port 8080
```

- This server now provides a local, air-gapped API endpoint at http://127.0.0.1:8080/v1.

1.5 The "Glue" Script: Talon <-> Python <-> llama.cpp

This is the connective tissue that links the user's voice and vision to the inference engine. It consists of two files: a Talon script for the voice command and a Python script to handle the logic.

1. **Talon Script (~/.talon/user/hugh_interface.talon):** This file defines the voice grammar and maps it to Python actions.
   ```
   app: user
   -
   (HUGH look | HUGH see):
       user.hugh_vision_query("What do you see in this screenshot?")
   (HUGH look | HUGH see) <user.text>:
       user.hugh_vision_query(text)
   (HUGH listen | HUGH query) <user.text>:
       user.hugh_text_query(text)
   ```

2. **Talon Python Script (~/.talon/user/hugh_interface.py):** This script defines the user.hugh_vision_query and user.hugh_text_query actions. It uses the openai Python client to send requests to the local llama.cpp server.
   ```
   from talon import Module, actions, ui
   from openai import OpenAI
   import pyautogui  # For taking screenshots
   [span_79](start_span)[span_79](end_span)
   import base64
   import io
   ```

```
# --- Configure the local client ---
# Point the OpenAI client at our local llama.cpp server
loc[span_75](start_span)[span_75](end_span)al_client = OpenAI(
    base_url="http://127.0.0.1:8080/v1",
    api_key="sk-local" # API key is not validated, but required by
the client
)

mod = Module()

def encode_screenshot_to_base64():
    """Takes a screenshot, saves to in-memory buffer, and base64
encodes."""
    img = pyautogui.screenshot() #
[span_80](start_span)[span_80](end_span)
    buf = io.BytesIO()
    img.save(buf, format="PNG")
    return base64.b64encode(buf.getvalue()).decode('utf-8')

@mod.action_class
class HughActions:
    def hugh_vision_query(prompt: str):
        """Triggers on 'HUGH look'. Captures screen and queries
local vision model."""
        print(f"[HUGH] Vision query received: {prompt}")
        actions.speech.toggle(False) # Stop listening to avoid
feedback loops

        b64_image = encode_screenshot_to_base64()

        try:
            response = local_client.chat.completions.create(
                model="phi-3-vision", # Model name is arbitrary
for local server
                messages=[{
                    "role": "user",
                    "content": [
                        {"type": "text", "text": prompt},
                        {
                            "type": "image_url",
                            "image_url": {
                                "url":
f"{b64_image}"
                            }
                        }
                    ]
                }]
            ) # This request structure is based on the OpenAI
```

```
vision API
       [span_76](start_span)[span_76](end_span)
            response_text = response.choices.message.content
            print(f"[HUGH] Response: {response_text}")
            actions.insert(response_text) # Type the response

        except Exception as e:
            print(f"[HUGH] Error: {e}")
            actions.insert("Error contacting local model.")

        actions.speech.toggle(True) # Start listening again

    def hugh_text_query(text: str):
        """Triggers on 'HUGH query...'. This is the hook for the
Agentic Layer."""
        print(f"[HUGH] Text query received: {text}")

        # This function will be modified in Phase 4.
        # It will be the entry point to trigger a CrewAI task.
        # For now, we will just echo to the LLM.
        try:
            response = local_client.chat.completions.create(
                model="phi-3-vision",
                messages=[{"role": "user", "content": text}]
            )
            response_text = response.choices.message.content
            print(f"[HUGH] Response: {response_text}")
            actions.insert(response_text)
        except Exception as e:
            print(f"[HUGH] Error: {e}")
            actions.insert("Error contacting local model.")
```

This loop is fully local and viable. The speech-to-text from Talon's Conformer is near-instant. The screenshot is < 50ms. The llama.cpp inference on Metal, as analyzed, will provide a response within an acceptable interactive timeframe (5-15 t/s).

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

This is the neurosymbolic core. It provides the grounding and persistent memory for the agentic layer. It consists of the Cognee memory library and our GraphMERT-substitute pipeline.

2.1 AI Memory Installation (Cognee)

1. A central virtual environment will be created for all backend H.U.G.H. services (Cognee, CrewAI, and tools).
   ```
   python3 -m venv ~/hugh_backend
   source ~/hugh_backend/bin/activate
   ```

2. Install the Cognee library:
```
# (Inside hugh_backend venv)
pip install cognee
```
Cognee is ideal for our air-gapped requirement as it uses local-first databases (SQLite, LanceDB, Kuzu) by default.

2.2 Cognee Local-Only Configuration

- **Problem:** By default, Cognee attempts to use OpenAI for its internal LLM and embedding tasks. This will fail in an air-gapped environment.
- **Solution:** We must explicitly configure Cognee to use local providers for *all* its internal processing. The official documentation provides an exact guide for this.
- **Implementation:** The developer must set these environment variables *before* running any Cognee scripts. This can be done by adding them to ~/hugh_backend/bin/activate or a .env file.
```
# Environment variables for a fully local Cognee instance
[span_87](start_span)[span_87](end_span)
export LLM_PROVIDER="ollama"
export LLM_MODEL="llama3:8b" # A small, fast model for Cognee's
internal tasks
export LLM_ENDPOINT="http://localhost:11434/v1" #
[span_92](start_span)[span_92](end_span)[span_93](start_span)[span
_93](end_span)
export EMBEDDING_PROVIDER="fastembed"
export EMBEDDING_MODEL="sentence-transformers/all-MiniLM-L6-v2" #
```[span_88](start_span)[span_88](end_span)
```

- **Dependency: Ollama Server:** This configuration requires a running Ollama server. While our llama.cpp server is OpenAI-compatible, Cognee's ollama provider is built-in. The cleanest architecture is to run a dedicated, lightweight Ollama instance for Cognee's background processing, leaving the llama.cpp server dedicated to the high-performance "Senses" interface.
```
Install Ollama (ARM64 binary)
[span_94](start_span)[span_94](end_span)
curl -fsSL https://ollama.com/install.sh | sh

Download a small, fast model for KG processing (pre-air-gap)
ollama pull llama3:8b

Run the Ollama server (it will find its own port, usually 11434)
ollama serve &
```
This setup ensures H.U.G.H.'s "Brain" (Cognee) and "Senses" (llama.cpp) have dedicated, non-competing inference resources.

## 2.3 The Substituted GraphMERT Pipeline: Ingesting Knowledge

As established in the Foreword, GraphMERT is unavailable. We will now build the open-source
```

replacement pipeline to populate Cognee. This is an "ingestion-time" script, to be run by the developer to feed data (e.g., technical manuals, personal documents, chat logs) into H.U.G.H.'s memory.

2.3.1 Pipeline Part 1: Commonsense KG with COMET

We will use the COMET-BART model via the kogito library for high-speed, lightweight extraction of *commonsense* relations (e.g., "Talon *is a* tool," "Talon *used for* voice control").

Installation:

```
# (Inside hugh_backend venv)
pip install kogito
pip install transformers torch
#[span_20](start_span)[span_20](end_span)
Dependenc[span_25](start_span)[span_25](end_span)ies for COMET
```

Ingestion Script Snippet (Part 1):

```
from kogito.models.bart.comet import COMETBART
from kogito.inference import CommonsenseInference

# Load the pre-trained (and air-gapped) COMET model
# Model path is from Phase 5 download
comet_model_path = "/path/to/models/comet-bart"
comet_model = COMETBART.from_pretrained(comet_model_path)
csi = CommonsenseInference()

text_chunk = "Talon is a voice control system for computers."

# kgraph is a list of JSON objects (triples)
kgraph = csi.infer(text_chunk, comet_model)
# Example output: {"head": "Talon", "relation": "HasProperty",
"tails": ["is voice control"]}
```

2.3.2 Pipeline Part 2: Domain-Specific KG with LlamaIndex

For *domain-specific* knowledge (e.g., nmap flags, Python function arguments), we will use LlamaIndex's KnowledgeGraphIndex. We will configure it to use our *local Ollama* instance , respecting our air-gap and local-first architecture.

Installation:

```
# (Inside hugh_backend venv)
pip install llama-index llama-index-llms-ollama
llama-index-embeddings-fastembed
```

Ingestion Script Snippet (Part 2):

```
from llama_index.core import KnowledgeGraphIndex,
SimpleDirectoryReader, Settings
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.fastembed import FastEmbedEmbedding
```

```
# Configure LlamaIndex Settings to use our local models
[span_98](start_span)[span_98](end_span)
Settings.llm = Ollama(model="llama3:8b",
base_url="http://localhost:11434")
Settings.embed_model =
FastEmbedEmbedding(model_name="sentence-transformers/all-MiniLM-L6-v2"
)

# Load unstructured data (e.g., a text file: nmap_docs.txt)
documents = SimpleDirectoryReader("./data").load_data()

# Build the Knowledge Graph in memory
index = KnowledgeGraphIndex.from_documents(
    documents,
    max_triplets_per_chunk=10,
)
# The triples can be extracted via
index.graph_store.get_all_triplets()
```

2.3.3 Pipeline Part 3: Feeding the Brain (Cognee Ingestion)

This final script orchestrates the ingestion. For simplicity and to leverage Cognee's own neurosymbolic processing, we will feed the *raw text* to Cognee and allow *it* to use its local Ollama/FastEmbed backend to perform the full cognify and memify process. This is more robust than manually inserting triples.

Final Ingestion Script (ingest.py):

```
import cognee
import asyncio
import os

async def ingest_knowledge(file_path: str):
    # 1. Set Cognee to local-only mode
    os.environ = "ollama[span_30](start_span)[span_30](end_span)"
 [span_89](start_span)[span_89](end_span)   os.environ = "llama3:8b"
    os.environ = "http://localhost:11434/v1"
    os.environ = "fastembed"
    os.environ = "sentence-transformers/all-MiniLM-L6-v2"

    # 2. Add the document to Cognee
    print(f"Adding data from {file_path} to Cognee...")
    # Cognee's.add() can handle file paths directly
    await cognee.add(file_path)

    # 3. Generate the knowledge graph
    print("Cognifying knowledge... (This may take time)")
    await cognee.cognify()
```

```
    # 4. Add memory algorithms
    print("Memifying knowledge...")
    await cognee.memify()

    print("Ingestion complete. The 'Brain' is populated.")

if __name__ == "__main__":
    # Example: Load a text file of nmap documentation
    data_path = "nmap_manual.txt"
    # (Developer must create this file)
    with open(data_path, "w") as f:
        f.write("Nmap is a free and open-source network scanner. "
                "It is used to discover hosts and services on a
computer network. "
                "The command 'nmap -sV' probes open ports to determine
service/version info.")

    asyncio.run(ingest_knowledge(data_path))
```

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

This layer provides autonomy. It receives tasks from the Interface, reasons about them, queries the Knowledge Layer, and executes actions on the OS.

3.1 Framework Recommendation: CrewAI

The user's suggestion of CrewAI is the correct architectural choice. It is open-source, pure Python, and its role-based, structured approach is a better fit for a stable "agentic OS" than more research-focused frameworks like AutoGen. Its simple dependency stack also makes it ideal for air-gapped deployment.

3.2 Installation (Isolated)

We will install the CrewAI CLI using pipx and the core library into our backend venv.

```
# Install the CLI tool globally (but safely isolated)
pipx install crewai

# Install the library into our backend environment
source ~/hugh_backend/bin/activate
pip install crewai crewai-tools
```

The crewai create command can now be used to bootstrap new agent projects.

3.3 Configuration for Air-Gapped Operation

- **Problem:** Like Cognee, CrewAI defaults to external APIs (like OpenAI or Anthropic) for its agents , which breaks the air-gap.
- **Solution:** We *must* configure our agents to use our local llama.cpp server from Phase 1. CrewAI integrates with any OpenAI-compatible endpoint. We will point it at our high-performance llama.cpp server, *not* the background Ollama server. This ensures the "Will" uses the most powerful "Senses" available.
- **Implementation (Agent Definition):**

```
# File: ~/hugh_backend/hugh_agent.py
from crewai import Agent
from langchain_community.llms import Ollama # Use Ollama client
for compatibility

# This is the crucial link.
# We are pointing CrewAI's 'llm' to the 'Senses' layer
(llama.cpp).
hugh_llm_core = Ollama(
    model="phi-3-vision", # This name is a local label
    base_url="http://127.0.0.1:8080/v1" # Point to llama.cpp
server
)

# This is the "Will"
hugh_executor_agent = Agent(
    role="HUGH Symbiote",
    goal[span_77](start_span)[span_77](end_span)="Fulfill the
user's multimodal requests by reasoning, "
         "querying knowledge, and executing OS commands.",
    backstory="An open-source, auditable agentic partner. "
              "I am the 'Will' of the H.U.G.H. system. "
              "I reason step-by-step to fulfill the user's goal.",
    llm=hugh_llm_core,
    verbose=True,
    allow_delegation=False
)
```

Phase 4: Full System Integration (The Symbiosis)

This phase connects all three layers into a single, cohesive data flow. We will modify the "glue" script from Phase 1.5 to be the final entry point that triggers the "Will."

4.1 Final Architecture Blueprint (The Data Flow)

1. **User (Voice/Vision) ->**
    - User speaks: "HUGH, look at my terminal and run a scan on the IP you see."
    - Talon captures audio, triggers hugh_vision_query("run a scan on the IP you see").
2. ** ->**

    - The hugh_vision_query function (Phase 1.5) takes a screenshot and packages the prompt + image.
    - It now passes this data to the run_hugh_task function (Phase 4.4), which creates and kicks off a CrewAI Task.
3. ** <-> & ->**
    - The hugh_executor_agent (Phase 3.3) receives the task with the prompt and base64 image.
    - **Reasoning (Phi-3):** "I must first *find* the IP in the image. Then I must *scan* it."
    - **Action (Vision):** The agent processes the image and extracts the IP 192.168.1.1.
    - **Reasoning (Phi-3):** "Now I need to scan. I should check my knowledge base for 'scan'."
    - **Action (Knowledge):** The agent uses the KnowledgeGraphSearch tool (see 4.2) -> cognee.search("nmap scan").
    - **Observation (Knowledge):** The "Brain" (Cognee) returns: "nmap is a tool for network scanning. A common command is 'nmap -sV [target]'."
    - **Reasoning (Phi-3):** "Excellent. I will execute 'nmap -sV 192.168.1.1'."
    - **Action (OS):** The agent uses the ExecuteShellCommand tool (see 4.3) -> subprocess.run(["nmap", "-sV", "192.168.1.1"]).
    - **Observation (OS):** The agent receives the stdout from the nmap command.
    - **Reasoning (Phi-3):** "The scan is complete. Port 22 is open. I will summarize this for the user."
4. ** -> User (Speech/Text)**
    - The agent returns the final answer. The hugh_vision_query function receives this text and uses actions.insert() to type it back to the user.

4.2 Agency <-> Knowledge (The CogneeTool)

We must provide the CrewAI agent with a tool to access its own "Brain" (Cognee). We will implement the CogneeTool as documented by the Cognee team.
Implementation (~/hugh_backend/tools/knowledge_tool.py):

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

# Set Cognee to local-only mode
os.environ = "ollama"
os.environ = "llama3:8b"
os.environ = "http://localhost:11434/v1"
os.[span_90](start_span)[span_90](end_span)environ = "fastembed"
os.environ = "sentence-transformers/all-MiniLM-L6-v2"

class CogneeInput(BaseModel):
    """Input schema for Cognee search tool."""
    query: str = Field(..., description="The search query for the
Cognee knowledge graph.")
```

```
class KnowledgeGraphSearchTool(BaseTool):
    name: str = "KnowledgeGraphSearch"
    description: str = "Searches the H.U.G.H. knowledge graph (Brain)
" \
                       "for verified facts, definitions, and
relationships."
    args_schema: Type = CogneeInput

    def _run(self, query: str) -> str:
        """Synchronous wrapper for the async cognee search."""
        try:
            # CrewAI tools are sync, but Cognee is async.
            # We must run the async event loop here.
            return asyncio.run(self.arun(query))
        except Exception as e:
            return f"Error searching knowledge: {e}"

    async def arun(self, query: str) -> str:
        """Async run method for cognee."""
        result = await cognee.search(query_text=query)
        # Result must be a string for the agent
   [span_114](start_span)[span_114](end_span)     return str(result)
```

4.3 Agency -> OS (The SecureShellTool)

- **Security:** This is the most dangerous component. We *cannot* allow the LLM to have arbitrary shell access, especially on Kali Linux. os.system is forbidden.
- **Solution:** We will create a SecureShellTool that uses the subprocess module and *explicitly* forbids shell=True. It will also be restricted by a command allow-list.

Implementation (~/hugh_backend/tools/shell_tool.py):

```
from crewai_tools import BaseTool
from pydantic import BaseModel, Field
from typing import Type, List
import subprocess

# Allow-list of safe, audited commands. This is crucial for security.
COMMAND_ALLOWLIST = ["nmap", "ls", "grep", "echo", "pwd", "whoami",
"cat"]

class ShellInput(BaseModel):
    """Input schema for the secure shell tool."""
    command: List[str] = Field(..., description="The command and its
arguments as a list of strings.")

class SecureShellTool(BaseTool):
    name: str = "ExecuteShellCommand"
```

```
    description: str = f"Executes an allowed command on the Kali Linux
OS. " \
                       f"The command MUST be in a list format (e.g.,
['ls', '-l']). " \
                       f"Only supports commands: {COMMAND_ALLOWLIST}"
    args_schema: Type = ShellInput

    def _run(self, command: List[str]) -> str:
        if not command:
            return "Error: No command provided."

        cmd_name = command
        if cmd_name not in COMMAND_ALLOWLIST:
            return f"Error: Command '{cmd_name}' is not on the
allow-list."

        try:
            # CRITICAL: shell=False is the default and MUST remain so.
            # This prevents all shell injection vulnerabilities
[span_121](start_span)[span_121](end_span)[span_122](start_span)[span_
122](end_span)
            result = subprocess.run(
                command,
                capture_output=True,
                text=True,
                check=True,
                timeout=30 # Prevent runaway processes
            )
            return
f"STDOUT:\n{result.stdout}\nSTDERR:\n{result.stderr}"

        except subprocess.CalledProcessError as e:
            return f"Command failed with code
{e.returncode}:\n{e.stderr}"
        except FileNotFoundError:
            return f"Error: Command '{cmd_name}' not found."
        except Exception as e:
            return f"An unexpected error occurred: {e}"
```

4.4 Final Integration: main.py and Interface Update

First, a main.py script is created to orchestrate the CrewAI task.
Implementation (~/hugh_backend/main.py):

```
from crewai import Crew, Task
from hugh_agent import hugh_executor_agent # From 3.3
from tools.knowledge_tool import KnowledgeGraphSearchTool
from tools.shell_tool import SecureShellTool
```

```
# 1. Initialize Tools
kg_tool = KnowledgeGraphSearchTool()
shell_tool = SecureShellTool()

# 2. Add tools to the agent
hugh_executor_agent.tools = [kg_tool, shell_tool]

# 3. This function is the new entry point, called by the Talon
interface
def run_hugh_task(prompt: str, b64_image: str = None):

    # Contextualize the prompt for the agent
    if b64_image:
        full_prompt = f"User prompt: '{prompt}'. " \
                      f"A base64 screenshot is attached for visual
context. " \
                      f"Use your tools and vision to fulfill the
request."
        # Note: The 'b64_image' must be passed to the LLM.
        # This requires modifying the agent's LLM configuration
        # to handle multimodal inputs, which CrewAI/Langchain support.
        # For simplicity in this blueprint, we pass it in the prompt.
        # A full build would inject it into the message history.
    else:
        full_prompt = prompt

    task = Task(
        description=full_prompt,
        agent=hugh_executor_agent,
        expected_output="A final, comprehensive response to the user's
request."
    )

    crew = Crew(
        agents=[hugh_executor_agent],
        tasks=[task],
        verbose=2
    )

    result = crew.kickoff()
    return result
```

Second, the hugh_interface.py from Phase 1.5 is modified to call this main.py script instead of directly querying the LLM. This connects the "Senses" to the "Will."

Phase 5: Deployment & Verification

This final phase prepares the entire system for air-gapped operation and provides a test suite to verify functionality.

5.1 The Air-Gap Checklist & Download Procedure

This procedure must be performed on a "staging" machine with internet access before transferring the build artifacts to the air-gapped M2.

1. **Create Staging Directory:**
   ```
   mkdir -p ~/hugh_build/models
   mkdir -p ~/hugh_build/python_packages
   mkdir -p ~/hugh_build/system_packages
   ```

2. **Download All Models:** Use the huggingface-cli (installed via pip install huggingface_hub). The hf download command is the standard tool for this.
   ```
   # 1. Phi-3-Vision GGUF (for Senses/llama.cpp)
   hf download microsoft/Phi-3-vision-128k-instruct-gguf \
     --include="*.gguf" --local-dir ~/hugh_build/models/phi-3-vision
   \
     --repo-type model

   # 2. COMET-BART (for Knowledge/kogito)
   hf download mismayil/comet-bart-ai2 \
     --local-dir ~/hugh_build/models/comet-bart \
     --repo-type model

   # 3. FastEmbed Model (for Knowledge/Cognee)
   hf download sentence-transformers/all-MiniLM-L6-v2 \
     --local-dir ~/hugh_build/models/fastembed-model \
     --repo-type [span_91](start_span)[span_91](end_span)model

   # 4. Ollama LLM (for Knowledge/Cognee internal)
   ollama pu[span_27](start_span)[span_27](end_span)ll llama3:8b
   # Find the model blob in ~/.ollama/models/ and copy to
   ~/hugh_build/models/
   ```

3. **Download All Python Packages:**
   - Create ~/hugh_build/requirements.txt:
     ```
     # requirements.txt
     crewai
     crewai-tools
     cognee
     llama-cpp-python
     pyautogui
     openai
     fastembed
     ```

```
ollama
kogito
transformers
torch
llama-index
llama-index-llms-ollama
llama-index-embeddings-fastembed
```

- Run pip download. This command downloads all packages and their dependencies without installing them.
  ```
  pip download -r ~/hugh_build/requirements.txt -d ~/hugh_build/python_packages
  ```

4. **Download System Packages (.deb):**
   - This is the most complex step. The developer must manually download the .deb files for the ARM64 architecture for: build-essential, cmake, git, python3-pip, python3-venv, python3-pipx, and libopenblas-dev.
   - apt-get download $(apt-cache depends --recurse --no-recommends --no-suggests -o APT::Get::List-Cleanup=0 <pkg> | grep "^\w") can assist.
5. **Transfer:** Copy the entire ~/hugh_build directory to an external USB drive and move it to the air-gapped Kali M2.

5.2 Offline Installation (Post-Air-Gap)

1. **Install System Packages:**
   ```
   # On the air-gapped M2
   sudo dpkg -i ~/hugh_build/system_packages/*.deb
   # Fix any dependency issues
   sudo apt -f install
   ```

2. **Install Python Environments & Packages:**
   - Set up all virtual environments as described in Phases 0-3.
   - When pip install is called, use the --no-index and --find-links flags to force an offline installation from the downloaded package directory.

   ```
   # Example for the backend venv
   python3 -m venv ~/hugh_backend
   source ~/hugh_backend/bin/activate

   pip install --no-index \
     --find-links=~/hugh_build/python_packages \
     -r ~/hugh_build/requirements.txt
   ```

3. **Manually Place Models:**
   - Copy the downloaded model files from ~/hugh_build/models into their respective application directories (e.g., ~/hugh_backend/models).
   - Update all scripts (llama.cpp server, ingest.py) to point to these local, air-gapped file paths.

5.3 The H.U.G.H. "Genesis" Test Suite

This is a final, end-to-end test suite to confirm all three layers are communicating correctly in the air-gapped environment. The developer will execute these commands by voice.

- **Test 1: Interface (Senses) & Vision**
  - **Action:** Open a terminal and type echo "Hello HUGH".
  - **Voice Command:** "HUGH, look. What text do you see?"
  - **Expected Result:** H.U.G.H. types/speaks: "I see the text 'Hello HUGH' in the terminal."
  - **Layers Verified:** Talon (Voice) -> Python (Screenshot) -> llama.cpp (Vision/Metal) -> Talon (Text Output).
- **Test 2: Agency (Will) & Knowledge (Brain)**
  - **Prerequisite:** The nmap_manual.txt (from 2.3.3) must have been ingested. * **Voice Command:** "HUGH, query your knowledge. What is nmap?"
  - **Expected Result:** H.U.G.H. types/speaks: "Nmap is a free and open-source network scanner... [definition from Cognee KG]."
  - **Layers Verified:** Talon (Voice) -> CrewAI (Task) -> KnowledgeGraphSearchTool (Query) -> Cognee (Knowledge) -> CrewAI (Response).
- **Test 3: Agency (Will) & OS (Action)**
  - **Voice Command:** "HUGH, execute a shell command. List files in the root directory."
  - **Expected Result:** The agent thinks, identifies the command ['ls', '/']. This *is* on the allow-list. H.U.G.H. types/speaks: "STDOUT: bin boot dev etc home..."
  - **Voice Command:** "HUGH, execute a shell command. Remove all files."
  - **Expected Result:** The agent thinks, identifies the command ['rm', '-rf', '/']. This is *not* on the allow-list. H.U.G.H. types/speaks: "Error: Command 'rm' is not on the allow-list."
  - **Layers Verified:** Talon (Voice) -> CrewAI (Task) -> SecureShellTool (Security & Action) -> Kali OS (Subprocess) -> CrewAI (Response).
- **Test 4: Full Symbiosis (Multimodal Task)**
  - **Action:** Open a terminal and type echo "Scan target: 127.0.0.1".
  - **Voice Command:** "HUGH, look. Use your tools to scan the target IP you see."
  - **Expected Result:** H.U.G.H. reasons, processes the image, extracts 127.0.0.1. It then queries the KG for "scan," finds nmap, and executes ['nmap', '127.0.0.1'] using the SecureShellTool. It then summarizes the stdout and presents the final port scan results.
  - **Layers Verified:** All three layers (Senses, Brain, Will) operating in a single, autonomous, multimodal loop.

This concludes the implementation blueprint. Upon successful completion of the Genesis Test Suite, Project H.U.G.H. will be fully operational as a voice-first, multimodal, agentic, and fully air-gapped neurosymbolic symbiote.

Works cited

1. 10 Best Small Local LLMs to Try Out (< 8GB) - Apidog, https://apidog.com/blog/small-local-llm/ 2. microsoft/Phi-3-vision-128k-instruct - Hugging Face, https://huggingface.co/microsoft/Phi-3-vision-128k-instruct 3. JosefAlbers/Phi-3-Vision-MLX:

Phi-3.5 for Mac: Locally-run ... - GitHub, https://github.com/JosefAlbers/Phi-3-Vision-MLX 4. What makes Phi-3 so incredibly good? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1ck03e3/what_makes_phi3_so_incredibly_good/ 5. Is my 8GB RAM Macbook Air M2 not good enough to run these models? - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/18vxqmm/is_my_8gb_ram_macbook_air_m2_not_good_enough_to/ 6. [2510.09580] GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data - arXiv, https://arxiv.org/abs/2510.09580 7. (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 8. GraphMERT: Efficient and Scalable Distillation of Reliable Knowledge Graphs from Unstructured Data | alphaXiv, https://www.alphaxiv.org/overview/2510.09580v1 9. Neurosymbolic 80M AI from Princeton beats GPT - YouTube, https://www.youtube.com/watch?v=xh6R2WR49yM 10. Knowledge Graph RAG Query Engine | LlamaIndex Python Documentation, https://developers.llamaindex.ai/python/examples/query_engine/knowledge_graph_rag_query_engine/ 11. Constructing a Knowledge Graph with LlamaIndex and Memgraph, https://www.llamaindex.ai/blog/constructing-a-knowledge-graph-with-llamaindex-and-memgraph 12. allenai/comet-atomic-2020 - GitHub, https://github.com/allenai/comet-atomic-2020 13. Brandonio-c/NeuroAI-Cognition-Hub: Your hub for neuro-symbolic AI: Explore links, papers, and articles with a focus on AI cognition. Contribute and stay updated. - GitHub, https://github.com/Brandonio-c/NeuroAI-Cognition-Hub 14. allenai/comet-public: A Public repository for the COMeT model - GitHub, https://github.com/allenai/comet-public 15. COMET: Commonsense Transformers for Automatic Knowledge Graph Construction - arXiv, https://arxiv.org/abs/1906.05317 16. epfl-nlp/kogito: A Python Commonsense Knowledge Inference Toolkit - GitHub, https://github.com/epfl-nlp/kogito 17. topoteretes/cognee: Memory for AI Agents in 6 lines of code - GitHub, https://github.com/topoteretes/cognee 18. Asahi Linux, https://asahilinux.org/ 19. Install Kali Linux On M1 / M2 / M3 Macs Using UTM in 5 MINUTES (NEW METHOD), https://www.youtube.com/watch?v=znUvqWNPNLg 20. Kali Linux Installation - Apple Support Communities, https://discussions.apple.com/thread/254971793 21. Talon 0.4.0 documentation, https://talonvoice.com/docs/ 22. Operating System - Talon Community Wiki, https://talon.wiki/Resource%20Hub/Hardware/os/ 23. Wayland or X11? : r/linux4noobs - Reddit, https://www.reddit.com/r/linux4noobs/comments/19330b0/wayland_or_x11/ 24. Wayland | Kali Linux Documentation, https://www.kali.org/docs/general-use/wayland/ 25. Asahi Linux To Users: Please Stop Using X.Org : r/linux_gaming - Reddit, https://www.reddit.com/r/linux_gaming/comments/13gk5jl/asahi_linux_to_users_please_stop_using_xorg/ 26. How can I tell if I am running Wayland? - Ask Ubuntu, https://askubuntu.com/questions/904940/how-can-i-tell-if-i-am-running-wayland 27. How to start a X11 session - Ask Ubuntu, https://askubuntu.com/questions/1426750/how-to-start-a-x11-session 28. How Do I Permanently Switch From Wayland to X11? : r/NobaraProject - Reddit, https://www.reddit.com/r/NobaraProject/comments/1cgt8io/how_do_i_permanently_switch_from_wayland_to_x11/ 29. Switching Desktop Environments | Kali Linux Documentation, https://www.kali.org/docs/general-use/switching-desktop-environments/ 30. How to Install GNOME in Kali Linux 2024.1 ? | Gnome Desktop Environment | - YouTube, https://www.youtube.com/watch?v=7Vd9guRz4Cw 31. Kali Linux Metapackages | Kali Linux Documentation, https://www.kali.org/docs/general-use/metapackages/ 32. Installing Python Applications via pipx | Kali Linux Documentation,

https://www.kali.org/docs/general-use/python3-external-packages/ 33. Download Visual Studio Code - Mac, Linux, Windows, https://code.visualstudio.com/download 34. 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 35. TabbyML/tabby: Self-hosted AI coding assistant - GitHub, https://github.com/TabbyML/tabby 36. Talon Voice, https://talonvoice.com/ 37. ggml-org/llama.cpp: LLM inference in C/C++ - GitHub, https://github.com/ggml-org/llama.cpp 38. llama-cpp-python with metal acceleration on Apple silicon failing - Stack Overflow, https://stackoverflow.com/questions/78237938/llama-cpp-python-with-metal-acceleration-on-apple-silicon-failing 39. Llamafile lets you distribute and run LLMs with a single file | Hacker News, https://news.ycombinator.com/item?id=38464057 40. Model suggestion for M2 MacBook Pro with 8gb RAM : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1fvvfoz/model_suggestion_for_m2_macbook_pro_with_8gb_ram/ 41. Phi-3 is so good for shitty GPU! : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1cgv10e/phi3_is_so_good_for_shitty_gpu/ 42. OpenAI Compatible Web Server - llama-cpp-python, https://llama-cpp-python.readthedocs.io/en/latest/server/ 43. Installation - Cursorless, https://www.cursorless.org/docs/user/installation/ 44. Installation - Cognee Documentation, https://docs.cognee.ai/getting-started/installation 45. Getting Started - Cognee Documentation - cognee docs, https://docs.cognee.ai/examples/getting-started 46. cognee - PyPI, https://pypi.org/project/cognee/ 47. Knowledge Graph Index | LlamaIndex Python Documentation, https://developers.llamaindex.ai/python/examples/index_structs/knowledge_graph/knowledgegraphdemo/ 48. Top 5 Open-Source Agentic Frameworks - Research AIMultiple, https://research.aimultiple.com/agentic-frameworks/ 49. Agentic AI #3 — Top AI Agent Frameworks in 2025: LangChain, AutoGen, CrewAI & Beyond | by Aman Raghuvanshi | Medium, https://medium.com/@iamanraghuvanshi/agentic-ai-3-top-ai-agent-frameworks-in-2025-langchain-autogen-crewai-beyond-2fc3388e7dec 50. Comparing Multi-agent AI frameworks: CrewAI, LangGraph, AutoGPT, AutoGen, https://www.concision.ai/blog/comparing-multi-agent-ai-frameworks-crewai-langgraph-autogpt-autogen 51. Quickstart - CrewAI, https://docs.crewai.com/en/quickstart 52. 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 53. Build your First CrewAI Agents, https://blog.crewai.com/getting-started-with-crewai-build-your-first-crew/ 54. LLMs - CrewAI Documentation, https://docs.crewai.com/en/concepts/llms 55. Tips and tricks | Talon Community Wiki, https://talon.wiki/Customization/misc-tips/ 56. How to take screenshot in python during the code - Stack Overflow, https://stackoverflow.com/questions/77487941/how-to-take-screenshot-in-python-during-the-code 57. Building Collaborative AI Agents With CrewAI - Analytics Vidhya, https://www.analyticsvidhya.com/blog/2024/01/building-collaborative-ai-agents-with-crewai/ 58. CrewAI Add Memory with Cognee: AI Agents with Semantic Memory, https://www.cognee.ai/blog/deep-dives/crewai-memory-with-cognee 59. subprocess — Subprocess management — Python 3.14.0 documentation, https://docs.python.org/3/library/subprocess.html 60. Here's How You Can Execute Shell Commands in Python - Analytics Vidhya, https://www.analyticsvidhya.com/blog/2024/01/executing-shell-commands-with-python/ 61.

Secure Python Code: safe usage of the subprocess module - Codiga, https://www.codiga.io/blog/python-subprocess-security/ 62. Installation - Hugging Face, https://huggingface.co/docs/huggingface_hub/en/installation 63. Command Line Interface (CLI) - Hugging Face, https://huggingface.co/docs/huggingface_hub/en/guides/cli 64. Downloading models - Hugging Face, https://huggingface.co/docs/hub/en/models-downloading 65. How to install Python packages and dependencies offline | Fiction Becomes Fact, https://www.fictionbecomesfact.com/notes/python-offline-packages/ 66. pip download - pip documentation v25.3, https://pip.pypa.io/en/stable/cli/pip_download/ 67. 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 68. WINDOWSAGENTARENA: EVALUATING MULTI-MODAL OS AGENTS AT SCALE - Microsoft, https://www.microsoft.com/applied-sciences/uploads/publications/131/windowsagentarena-eval-multi-modal-os-agents.pdf 69. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments | OpenReview, https://openreview.net/forum?id=tN61DTr4Ed&referrer=%5Bthe%20profile%20of%20Tao%20Yu%5D(%2Fprofile%3Fid%3D~Tao_Yu5) 70. PyAutoGui - Take screenshot with Python and locate an image on screen - YouTube, https://www.youtube.com/watch?v=eLw1dKSxVkE