I am building an llm-based Jarvis bot on my Mac that will allow me to achieve any task that I ask it to while also being able to have a conversation with me. These include any tasks for which an AppleScript or an automator workflow or a python or shell script can be made. Here is my workflow pipeline: Use whisper.cpp for STT -> parse text to LLM -> LLM converts text to structured json -> parse json -> function that runs corresponding AppleScript/automator workflow/python/bash script. I have made a MVP. I run the code using python3 assistant.py. I can say “delete test.txt from desktop”, voice is transcribed to text, converted to JSON: {"action": "delete\_file", "params": {"path": "~/Desktop/test.txt"}} which can be used to execute the delete\_file.scpt. I have an Operator project folder with the following files: /Operator/assistant.py, /Operator/executor.py, /Operator/llama.cpp/models/Mistral-7B-Instruct-v0.3.Q4\_K\_M.gguf, /Operator/whisper.cpp/models/ggml-base.en.bin, /Operator/scripts/delete\_file.scpt.

I want to implement the tasks below but I am not sure where to start. Think deeply and help me figure out where and how to start by providing me with a detailed roadmap.

Here are the tasks I want to achieve in no particular order:

-Fine-tune the model using QLORA to output structured JSON (I am not sure when to do this since, in order to fine-tune it, I need a dataset of structured JSON but I have not made all the scripts and functionality plugins so I want to fine-tune only once I am satisfied with all the scripts and workflows I have added so that I know the kind of structured json I want the model to create for the many tasks).

- Automate my workflow using node-red for simpler testing and execution.

-Be able to type in the task as well (currently can only speak and transcribe text, should be able to type text directly as well and parse to LLM).

- I have a bunch of task plugins in the form of python files in addition to the scripts and workflows I want to use. I need to know how to implement them and when to do so.

- Implement the Jarvis’s personality aspect. I should be able to get Jarvis talk back to me. He should also be able to detect when I am asking to execute a task vs when I’m just having conversation with him (probably using JSON with action parameter: respond or something similar).

-Have some kind of memory to know what he's done previously, for ex: deleted a file, should be able to refer back and undo the task along with other operations.

-Eventually implement some kind of GUI for the user and convert the project into app.

- Add more scripts and workflows for the following comprehensive functionality list:

System Operations

\* File management (create/delete/copy/move/compress)

\* Process management (start/stop/monitor)

\* System controls (sleep/restart/shutdown/lock)

\* Clipboard management

\* Display controls (brightness/resolution)

Automation & Productivity

\* Calendar management

\* Email operations

\* Reminder system

\* Meeting scheduling

\* Document automation (create PDFs/merge files)

Network & Connectivity

\* WiFi management

\* Network diagnostics

\* VPN controls

\* SSH tunnel management

\* Port monitoring

Media Control

\* System volume control

\* Music playback

\* Video processing

\* Screenshot capture

\* Screen recording

Development Operations

\* Git operations

\* Container management (Docker)

\* Server management

\* Code compilation

\* API testing

Web Interactions

\* Browser control

\* Web scraping

\* Social media interactions

\* Stock monitoring

\* News aggregation

AI & Advanced Features

\* Real-time translation

\* Content summarization

\* Sentiment analysis

\* Data visualization

\* Machine learning model execution

Here is my code:

Operator/assistant.py:

# This script is the main interface for the voice assistant. It records audio from

# the microphone, transcribes it, processes it with the LLaMA model, and executes

# the resulting action using the executor module. The main\_loop function repeatedly

# records, transcribes, processes, and executes commands until the user quits. The

# record\_voice function uses SoX to record audio from the system microphone. The

# transcribe\_audio function uses whisper.cpp to transcribe the recorded audio into text.

# The process\_command function feeds the user's text command into the LLaMA model with

# a system prompt, expecting a JSON output specifying {"action": "...", "params": {...}}.

# The main interface for the assistant is the main\_loop function, which orchestrates the

# entire process. It records audio, transcribes it, processes it with LLaMA, and executes

# the resulting action until the user quits.

#

import os

import subprocess

from pydub import AudioSegment

import json

import time

from executor import execute\_action

from llama\_cpp import Llama

# -------------------------------------------------------------------

# 1. GLOBAL SETUP

# -------------------------------------------------------------------

# Path to your LLaMA-based model. Adjust as necessary.

MODEL\_PATH = "./llama.cpp/models/Mistral-7B-Instruct-v0.3.Q4\_K\_M.gguf"

# You can tweak these parameters to find the right speed vs. quality tradeoff.

llm = Llama(

model\_path=MODEL\_PATH,

n\_ctx=512, # smaller context speeds up inference for short commands

n\_threads=8,

n\_gpu\_layers=999, # (Uncomment if you want partial GPU acceleration on Apple Silicon)

n\_batch=256 # (Helps performance for short queries in many cases)

)

SYSTEM\_PROMPT = """Convert user commands into a JSON object with the form:

{

"action": "<script\_name>",

"params": {

"key": "value",

...

}

}

Available actions (scripts): delete\_file, move\_file, create\_folder, open\_file, search\_files.

Example: "move/delete test.txt to Trash/Recycle Bin/Garbage":

Output: {"action": "delete\_file", "params": {"path": "~/Desktop/test.txt"}}

Example: "Move report.pdf from Downloads to Documents":

Output: {"action": "move\_file", "params": {"source": "~/Downloads/report.pdf", "destination": "~/Documents"}}

"""

def record\_voice(filename="command.wav", duration=3):

"""

Records audio from the system microphone for `duration` seconds.

Make sure you have SoX installed (brew install sox).

"""

print(f"Recording {duration} second(s) of audio... speak now.")

os.system(f"sox -d -r 16000 -c 1 {filename} trim 0 {duration}")

def transcribe\_audio():

"""

Uses whisper.cpp to transcribe the recorded audio into text.

"""

if not os.path.exists("command.wav"):

raise FileNotFoundError("Record audio first using record\_voice()")

try:

# Convert audio to 16kHz, mono, 16-bit PCM

sound = AudioSegment.from\_file("command.wav")

sound = (sound.set\_frame\_rate(16000)

.set\_channels(1)

.set\_sample\_width(2))

sound.export("command\_16k.wav", format="wav", codec="pcm\_s16le")

# Paths

whisper\_cli\_path = os.path.abspath("whisper.cpp/build/bin/whisper-cli")

# You can speed up transcription by switching to "ggml-tiny.en.bin" or "ggml-small.en.bin"

# if you have them downloaded in the models folder.

model\_path = os.path.abspath("whisper.cpp/models/ggml-base.en.bin")

audio\_path = os.path.abspath("command\_16k.wav")

# Run whisper.cpp CLI with multi-threading (-t 8)

print("Transcribing audio with whisper.cpp (this may take a few seconds)...")

result = subprocess.run([

whisper\_cli\_path,

"-m", model\_path,

"-f", audio\_path,

"-otxt",

"-l", "en",

"-t", "8"

], capture\_output=True, text=True)

if result.stderr:

print("Whisper CLI stderr:", result.stderr)

txt\_path = audio\_path + ".txt"

if not os.path.exists(txt\_path):

raise FileNotFoundError(f"Missing Whisper output file: {txt\_path}")

with open(txt\_path, "r") as f:

return f.read().strip()

except Exception as e:

print(f"Transcription failed: {str(e)}")

return ""

def process\_command(text):

"""

Feeds the user's text command into the LLM with a system prompt,

expecting a JSON output specifying {"action": "...", "params": {...}}.

"""

prompt = f"{SYSTEM\_PROMPT}\n\nUser: {text}\nAssistant:"

print("Sending to LLM:", prompt)

# You can experiment with smaller max\_tokens if you prefer.

output = llm(prompt, max\_tokens=128, temperature=0.2, stop=["\n"], echo=False)

# LLM output (the raw text after "Assistant:")

json\_str = output["choices"][0]["text"].strip()

print("LLM Raw Output:", output)

print("Attempted JSON:", json\_str)

# Validate JSON

try:

# Just to ensure it's valid JSON. We'll pass the raw string forward to executor.

\_ = json.loads(json\_str)

return json\_str

except json.JSONDecodeError:

print(f"Invalid JSON from LLM: {json\_str}")

return ""

def main\_loop():

"""

Repeatedly record, transcribe, process with LLM, and execute until user quits.

"""

while True:

user\_input = input("\nPress Enter to record a command (or type 'q' to quit): ").strip().lower()

if user\_input == 'q':

print("Exiting...")

break

# 1. Record from microphone

record\_voice()

# 2. Transcribe the audio

text\_command = transcribe\_audio()

print("Transcription:", text\_command)

# 3. Convert to JSON command

json\_output = process\_command(text\_command)

# 4. Execute

if json\_output:

execute\_action(json\_output)

if \_\_name\_\_ == "\_\_main\_\_":

main\_loop()

Operator/executor.py:

# This script is responsible for executing the actions specified by the assistant. It loads scripts from the scripts folder, executes

# them with the provided parameters, and returns the result. The execute\_action function takes a JSON string as input, extracts the action

# and parameters, and executes the corresponding script. It uses subprocess to run the scripts and returns True if successful, False otherwise.

# The load\_scripts function scans the scripts folder for available scripts and maps them to actions based on the script filename. The execute\_script

# function takes an action and parameters, finds the corresponding script, expands user paths, and executes the script with the parameters as arguments.

# The main interface for the assistant is the execute\_action function, which processes the JSON input, extracts the action and parameters, and calls

# execute\_script to run the corresponding script. If an error occurs during execution, it returns False.

import os

import json

import subprocess

# ------------------------------------------------------------------------------

# 1. Dynamically load scripts from the 'scripts' folder

# The 'action' name is the script's base filename (no extension).

# ------------------------------------------------------------------------------

SCRIPTS\_DIR = os.path.join(os.path.dirname(\_\_file\_\_), "scripts")

def load\_scripts():

"""

Scans the SCRIPTS\_DIR for any files (e.g. 'delete\_file.scpt', 'move\_file.sh')

and maps them to actions, e.g. 'delete\_file' -> full path to script.

"""

script\_map = {}

if os.path.isdir(SCRIPTS\_DIR):

for script\_file in os.listdir(SCRIPTS\_DIR):

# Ignore hidden/system files

if script\_file.startswith("."):

continue

action\_name, ext = os.path.splitext(script\_file)

full\_path = os.path.join(SCRIPTS\_DIR, script\_file)

script\_map[action\_name] = full\_path

return script\_map

ACTION\_SCRIPTS = load\_scripts()

# ------------------------------------------------------------------------------

# 2. Execute a script by name + pass in the params as argv

# ------------------------------------------------------------------------------

def execute\_script(action, params):

"""

Given an action (e.g. "delete\_file") and a params dict

(e.g. {"path": "~/Desktop/test.txt"}),

find the corresponding script in ACTION\_SCRIPTS and run it.

Pass the params as command-line arguments, in the order they appear in the dict.

"""

script\_path = ACTION\_SCRIPTS.get(action)

if not script\_path:

print(f"No script found for action '{action}'")

return False

# Expand user (~) in all param values

expanded\_params = [os.path.expanduser(str(value)) for value in params.values()]

\_, ext = os.path.splitext(script\_path)

if ext == ".scpt":

# AppleScript -> osascript

cmd = ["osascript", script\_path] + expanded\_params

elif ext == ".sh":

# Shell script

cmd = ["bash", script\_path] + expanded\_params

elif ext == ".py":

# Python script

cmd = ["python3", script\_path] + expanded\_params

else:

# Fallback to trying to execute directly (chmod +x might be needed)

cmd = [script\_path] + expanded\_params

print("Executing:", cmd)

try:

subprocess.run(cmd, check=True)

return True

except subprocess.CalledProcessError as e:

print(f"Script execution failed: {str(e)}")

return False

# ------------------------------------------------------------------------------

# 3. Main interface for the assistant

# ------------------------------------------------------------------------------

def execute\_action(json\_str):

try:

command = json.loads(json\_str)

action = command.get("action")

params = command.get("params", {})

if action:

return execute\_script(action, params)

else:

print("JSON missing 'action' field.")

except Exception as e:

print(f"Error in execute\_action: {str(e)}")

return False

Operator/scripts/delete\_file.scpt:

on run argv

set filePath to POSIX file (item 1 of argv)

tell application "Finder"

delete filePath

end tell

end run