ChatGPT

Building a Portable Ultron Assistant (Windows .exe) with Offline-First AI

Overview & Objectives

Ultron is being transformed into a **portable Windows executable** that can run directly from a USB drive without installation. The goals are: - Use an existing `ultron_config.json` (on the USB, e.g. `D:\ULTRON`) at runtime to load sensitive keys (like the OpenAI API key) rather than hard-coding them. - Implement **offline-first AI logic** – the assistant should work without internet by using a local language model, while still allowing OpenAI API calls when available (i.e. **fallback from OpenAI to a local model**). - Provide a simple **GUI interface** to show task progress (with progress bars, timers, status labels, etc.) for better user feedback. - Ensure **portable path handling** so all file/model references are relative to the USB (no absolute paths or system dependencies). - **Sync logs and configs to Google Drive** automatically when the drive is available (e.g. `H:\My Drive\ultron`). - Make the design extensible for future features (e.g. voice input/output). - Package the entire application as a **single executable (.exe)** via PyInstaller, and prepare the GitHub repo `dqikfox/ultron` with full source, build scripts, and documentation.

By the end, Ultron will be a self-contained assistant that you can plug-and-play on any Windows machine, with offline AI capabilities and cloud backup of its data.

Project Structure & Config Management

Begin by organizing the project files logically in the `ultron` repository. A suggested structure:

```
ultron/
├─ ultron_main.py        # Main entry point for the app (initializes GUI and
logic)
├─ ultron_ai.py          # Module for AI logic (OpenAI API calls and local
model interface)
├─ ultron_gui.py         # Module for GUI components and event loops
├─ ultron_config.json    # Config file (NOT tracked in git if it contains
sensitive keys)
├─ requirements.txt      # Python dependencies (openai, gpt4all, etc.)
├─ build.spec            # (Optional) PyInstaller spec file for fine-tuned
build
└─ README.md             # Documentation & usage instructions
```

Config File (`ultron_config.json`): This file (located on the USB in `D:\ULTRON`) will store user-specific settings such as the OpenAI API key, and possibly preferences (like which local model to use, or toggling online/offline mode). By reading this at runtime, we avoid hardcoding keys. For example, the JSON might look like:

```
{
  "openai_api_key": "sk-...yourkey...",
  "preferred_local_model": "orca-mini-3b.ggmlv3.q4_0.bin",
  "use_openai": true   // or a flag to prefer online if available
}
```

The code should load this config at startup. In `ultron_main.py`:

```
import json, os
config_path = os.path.join(os.path.dirname(__file__), 'ultron_config.json')
with open(config_path, 'r') as cf:
    config = json.load(cf)
OPENAI_API_KEY = config.get("openai_api_key")
```

Note: The `ultron_config.json` itself will reside on the USB; you might **not** check it into GitHub (especially if it contains a real API key). Instead, include a `ultron_config_template.json` in the repo with dummy values and instruct the user to rename it and insert their key.

Handling Relative Paths for Portability

To ensure the program uses files relative to its location (whether running as source or as a frozen exe), use a runtime check to find the application's directory. Python's `sys` module can tell us if we're running from a PyInstaller bundle. A common pattern is:

```
import sys, os
if getattr(sys, 'frozen', False):
    # Running as bundled .exe
    base_path = os.path.dirname(sys.executable)
else:
    # Running from source .py
    base_path = os.path.dirname(__file__)
```

This gives `base_path` as the folder where the `ultron_main.py` or `ultron.exe` resides. We then construct file paths from this base. For example, to load the config or model:

```
config_path = os.path.join(base_path, "ultron_config.json")
model_path = os.path.join(base_path, "models", "local_model.bin")
```

This approach was recommended for PyInstaller executables: it checks if the app is a script or frozen exe, then uses `sys.executable`'s path in the frozen case [1]. By doing this, **Ultron will always look for files in its own directory (the USB drive)**, not an absolute path like `C:\Ultron`. This makes it fully portable; you can run it from **any drive letter** or folder.

Log File Paths: Similarly, if Ultron writes logs (e.g. `ultron_log.txt` or voice input logs), have it write to `os.path.join(base_path, "ultron_log.txt")` so that the log stays on the USB by default. This way, all outputs stay with the application.

Offline-First AI Logic (Local Model + OpenAI Fallback)

One of the core enhancements is enabling Ultron to function without internet by using a local Large Language Model (LLM). We will integrate an open-source model and use the OpenAI API only as a secondary option.

Choosing a Local Model: For portability and offline use, a smaller **GPT4All** or **LLaMA-based** model is suitable. GPT4All provides a convenient Python API to run local models on CPU. For example, the GPT4All library can load models like `"orca-mini-3b"` (around 2GB) or others in the 3-8GB range that can run on a typical laptop CPU [2] [3]. You might distribute one of these model files on the USB (perhaps under a `models/` directory).

Integrating GPT4All: Install the `gpt4all` Python package and include it in `requirements.txt`. Using it looks like:

```
from gpt4all import GPT4All
model = GPT4All(model_name="orca-mini-3b.ggmlv3.q4_0.bin",
model_path=base_path)  # load from USB path
```

We specify `model_path=base_path` (or the `models` subfolder) to tell GPT4All where to find the model file. The GPT4All documentation confirms we can set a custom model directory or file path [4]. We should also **disable auto-download** in case the model file isn't found – since we want true offline operation. This can be done by `GPT4All(..., allow_download=False)` [5]. That way, if the model is missing, the program can alert the user rather than trying to fetch it online.

Local vs OpenAI Workflow: Implement the logic such that whenever Ultron needs to generate a response: 1. **Try using OpenAI** (if allowed by config and presumably online). Use the OpenAI Python API with the key from config:

```
import openai
openai.api_key = OPENAI_API_KEY
try:
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[...],
        timeout=5  # short timeout
    )
    answer = response['choices'][0]['message']['content']
except Exception as e:
    answer = None  # Will fallback to local
```

Here we attempt a quick API call. If the network is unavailable or any error occurs, we catch it (setting `answer=None`). 2. **If OpenAI failed**, or if we are in forced offline mode, use the **local model**:

```
if answer is None:
    # Fallback to local LLM
    answer = model.generate(prompt)
```

The `model.generate()` will produce a completion from the local model. Depending on the GPT4All model, you might want to wrap this in a simple prompt template (some models expect an assistant style prompt).

1. Return or display the `answer` in the UI.

Using this strategy, Ultron will **prefer offline** by default (since if offline, it just goes straight to local). Even if `use_openai=true` in config, the code should gracefully handle the absence of internet by switching to the offline model automatically. If you want true *"offline-first"* behavior (always try local first for privacy/cost reasons, and only use OpenAI if local result is insufficient), you could invert the logic: generate with local model first, and only if a certain condition (like confidence or response quality) is not met, then call OpenAI. However, that's harder to quantify automatically. The simpler interpretation is: **attempt OpenAI if allowed, otherwise use local** – which covers the offline scenario as a fallback.

Note on Local Model Performance: Keep in mind local models (especially 3B-8B param ones) will be slower and possibly less capable than OpenAI's API. You may want to restrict the length of responses or use smaller prompts to maintain responsiveness. The GUI's progress indicators (discussed below) will help show that something is happening during the potentially longer processing time of the local model.

GUI Implementation (Progress Indicators & Status)

A basic GUI will make Ultron much more user-friendly, providing visual feedback on what it's doing. We don't need a very complex UI – just a window that might contain: - **Status Labels**: e.g. "Idle", "Listening for input...", "Contacting OpenAI API...", "Generating response...", etc. This gives a textual indication of the current operation. - **Progress Indicators**: For tasks that take noticeable time (downloading model on first run, loading the model into memory, waiting for AI response), a progress bar or spinner is helpful. For example, if using speech recognition (in future) or reading a file, show a progress bar for that duration. For text generation, if using streaming token output, you could animate a progress bar or simply show a busy indicator. - **Timers**: Possibly a small timer label to show how long the current task has been running (for debugging or user patience). This could reset each time a new operation starts. It's optional, but since the prompt specifically mentions timers, you might show e.g. "Time Elapsed: 12s" for the current action. - **User Controls**: Depending on how Ultron is triggered, the GUI could include a button to start/stop listening (for voice commands), or a field to manually input a query, etc. Initially, if Ultron was monitoring a folder for triggers, you might not need many buttons – but for completeness, a "Quit" button or a "Run Task" button could be present.

Choice of GUI Framework: Python's built-in **Tkinter** is a good choice for portability (no extra dependencies, works with PyInstaller nicely). It can create simple windows, labels, and progress bars (using `ttk.Progressbar`). Another option is **PyQt5/PySide6** for a more advanced GUI, but that will increase the size and complexity (and PyInstaller bundling becomes heavier). For our needs (just a few labels and bars), Tkinter should suffice.

Implementing the GUI: In `ultron_gui.py`, you might use a class to create a `tk.Tk()` window:

```
import tkinter as tk
from tkinter import ttk

class UltronGUI:
    def __init__(self):
```

```
        self.root = tk.Tk()
        self.root.title("Ultron Assistant")
        # Create UI elements
        self.status_label = tk.Label(self.root, text="Status: Idle")
        self.status_label.pack(padx=10, pady=5)
        self.progress = ttk.Progressbar(self.root, length=200,
mode="indeterminate")
        # Initially, keep progress bar indeterminate (spinning) during tasks.
        self.progress.pack(padx=10, pady=5)
        self.timer_label = tk.Label(self.root, text="Elapsed: 0s")
        self.timer_label.pack(padx=10, pady=5)
        # Possibly more UI elements like a start/stop button if needed.
    def update_status(self, text):
        self.status_label.config(text=f"Status: {text}")
        self.root.update_idletasks()
    def start_progress(self):
        self.progress.start(10)  # start spinning (10ms step)
    def stop_progress(self):
        self.progress.stop()
    # ... (and so on)
```

The above is a simple pattern: you update the label text to reflect current status, and you can start/stop the progress bar's animation. The timer can be updated using a repeating `after()` call in Tkinter to tick every second.

Threading Consideration: It's important not to block the GUI main loop while doing AI processing. If you call OpenAI API or the local model in the same thread as Tkinter's mainloop, the window will freeze until the response is done. To avoid this, you should perform AI calls in a **worker thread** or use `asyncio` if using an async library. The typical approach: - When a command is triggered (say, user presses a button or a file appears), you spawn a `threading.Thread` to handle the AI response generation. - In the GUI, immediately update status to "Processing..." and start the progress indicator. - In the background thread, do the heavy work (OpenAI call or local model `.generate()`). - Once done, have the thread update the GUI with the result. *Note:* Direct GUI updates from a background thread aren't safe; instead use thread-safe mechanisms. For example, Tkinter allows `.after()` scheduling: the worker thread can put the result in a queue or a shared variable, and then call `gui.root.after(0, gui.display_result, result)` to safely transfer control back to the main thread to update GUI elements (like showing the answer, stopping the progress bar, etc.).

Displaying the Result: If Ultron's response is mainly text, you could either pop it up in a message box or show it in a text widget in the GUI. For now, a multiline `tk.Text` box that appends "User: ... \nUltron: ..." could serve as a simple chat log. This would also provide some **history** of interaction within the session. If voice output is to be added later, you can still show the text while speaking it.

Example Workflow in GUI: 1. Ultron is idle (GUI says "Idle", progress bar not moving). 2. When a trigger occurs (e.g. file creation or user click "Ask"), update status to "Transcribing audio..." or "Reading input..." if needed. (Progress bar could run if this takes time). 3. Then update status to "Querying OpenAI..." or "Generating response offline..." depending on the path. Start an indeterminate progress bar spin to indicate work. 4. As soon as a response is ready, stop the progress bar, maybe flash status "Done" and then back to "Idle". Display the answer in the text area. 5. The timer label can show how long step 3 took, or overall round-trip time.

This kind of feedback loop greatly improves usability, as the user knows the system is working and not just hung.

Packaging with PyInstaller (Build Script)

With the application code and GUI ready, we use **PyInstaller** to compile everything into an `.exe`. You can create a build script or just use the command-line. The key is to ensure all dependencies are included and paths are handled.

PyInstaller Command: In the simplest form, from the project directory run:

```
pyinstaller --onefile --windowed ultron_main.py
```

Here: - `--onefile` packs it into a single EXE file [6]. - `--windowed` (or `-w`) means no console window will open (since we have a GUI). This is appropriate for a GUI app so the user doesn't see a blank console behind the GUI. - You might also add `--icon=ultron.ico` if you have an icon file for the app (optional).

PyInstaller will create a `dist/ultron_main.exe` (or just `ultron_main.exe` in `dist` folder). This is the portable executable you can copy to the USB drive (e.g., put it in `D:\ULTRON\ultron.exe`).

Including Data Files: Because we **want to use external files (config, model, logs)** from the same folder, we do **not** need to bundle those into the exe. In fact, it's better not to package the model or config in the exe – keep them as separate files on the USB. This way, the config (with API key) can be edited by the user easily, and the model (which might be several GB) isn't baked into the EXE (which would make it huge and slow to load). So the PyInstaller command above is fine (it will include the Python code and necessary libraries).

If you had additional data like a default config or some resource files that the code expects to be embedded, PyInstaller's `.spec` file or `--add-data` option could be used. For instance, if you have a GUI layout file or a small default JSON, you'd add `--add-data "path\to\file;dest_folder"`. But for our case, it's not needed – the external files will be accessed via relative paths at runtime.

Testing the EXE: After building, test it on a clean Windows machine (or at least outside your development environment) to ensure it runs without missing dependencies. The GUI should launch and not throw import errors. Common pitfalls to watch for: - Some libraries like `openai` or `gpt4all` might require additional hidden imports. If the exe fails due to missing modules, you may need to add them in the spec or use `--hidden-import`. For example, GPT4All might use ctypes or DLLs; PyInstaller usually grabs these, but be attentive. - The local model file should be present in the expected relative location, otherwise the program will log an error (you can handle this by checking file existence and showing a message like "Local model not found. Please ensure `models/xyz.bin` is in the Ultron folder.").

Once it's confirmed working, you have the **portable Ultron.exe** ready.

Build Script/Instructions: It's a good idea to include in the GitHub repo: - A `build_instructions.md` or a section in the README detailing how to install PyInstaller and run the

build. - Possibly a `ultron.spec` file (PyInstaller spec) if custom configurations are needed, though for now the command-line options suffice.

In the GitHub `dqikfox/ultron` repo, add all source files (`.py` files, README, requirements.txt). The OpenAI API key should **not** be in the source code or repo; it stays in the `ultron_config.json` on the USB (which is in `.gitignore` to avoid committing it). Provide an example config without the key.

Also include usage instructions in the README: - How to run `ultron.exe` from the USB (simply double-click, or run via command line). - Mention that it expects certain files (config, model) in the same folder. - If applicable, instructions on how to obtain a local model (e.g. download link for a recommended GPT4All model) and where to place it (e.g. in a `models/` subfolder). - Describe basic usage, e.g. "Ultron will monitor the folder for voice input or accept text input via the GUI... (depending on how the interaction is designed)."

By making these instructions clear, anyone who pulls the repo or uses the USB will understand how to set up and use Ultron.

Google Drive Sync (Logs & Config Backup)

To automatically **sync logs and configs to Google Drive**, Ultron should detect if the Google Drive is accessible on the host system. The user indicates their Google Drive is mounted as `H:\My Drive\ultron`. This suggests that on Windows, Google Drive's "My Drive" appears as drive `H:` with a folder `ultron` inside it (the user likely created an `ultron` folder in their Google Drive).

The strategy: - On startup (or on shutdown, or periodically), check if the path `H:\My Drive\ultron` exists on the machine. You can do:

```
gdrive_path = r"H:\My Drive\ultron"
if os.path.isdir(gdrive_path):
    have_gdrive = True
else:
    have_gdrive = False
```

- If available, perform sync. "Sync" in this context likely means copying any new or changed files from the USB to that Google Drive folder. Specifically, **logs** (like `ultron_log.txt`) and **config**. Since the config contains the API key, you might want to back it up to cloud for safety – but be cautious (it's sensitive data). If the Google Drive is your personal account, it should be fine.

A simple one-way backup approach: whenever Ultron writes to its log file, also write (append) to a log in the Google Drive folder. Or at least at the end of the session, copy the whole log file over:

```
import shutil
if have_gdrive:
    try:
        shutil.copy(os.path.join(base_path, "ultron_log.txt"),
                    os.path.join(gdrive_path, "ultron_log_backup.txt"))
```

```
    except Exception as e:
        print("Warning: Could not sync log to Google Drive:", e)
```

Likewise for the config: if the config is updated or at least on start, you could copy `ultron_config.json` to `H:\My Drive\ultron\ultron_config_backup.json`. This ensures that even if the USB is lost, the config (key) and logs are stored in your Google Drive safely.

For a more robust sync (two-way or real-time), a more complex approach or use of the Google Drive API would be needed. But given the scope, the **simple backup on availability** approach is likely sufficient: - Check on program exit (or periodically every X minutes) if Google Drive is present, and then update the files on Google Drive. - If needed, also consider updating the local files from Google Drive if they are newer (for example, if you ran Ultron on another machine and the log on Google Drive has new info, you might want to merge or keep an archive). This can be complex; it might be easier to treat the Google Drive copies as backups only, not the primary source. So we will assume one-way sync from local->Drive for logs. For config, since it rarely changes, backing it up is fine (if you manually update config on Google Drive for some reason, you'd have to copy it to USB manually at this point).

Implementing the sync should be done carefully to not hang the main app. It can be done in a background thread as well, or very quickly at start/end: - At startup: if GDrive available, maybe load config from either local or GDrive. But since user specifically said use the local `D:\ULTRON\ultron_config.json` at runtime, we stick with that as the source of truth. - At exit: attempt to copy files to GDrive. This won't affect the main operation and can be done quickly.

Be sure to handle exceptions (like if GDrive path is protected or no write permission) and maybe log them.

In the **README instructions**, mention that *"Ultron will copy its log and config to Google Drive (H:\My Drive\ultron) if available, for backup. Ensure you have Google Drive for Desktop running and that the drive letter is correct (H:)."* This informs the user about the feature and any requirements.

Future Voice Support Considerations

We are planning to **extend voice support** later, so it's wise to design the system in a modular way to accommodate that: - **Voice Input (STT)**: Likely you will integrate a speech-to-text component. This could be an offline library like Vosk (for offline STT) or using OpenAI Whisper API if online. The architecture can have a module (say `ultron_voice.py`) that handles capturing microphone input and converting to text. This module can run a listening loop (possibly in another thread or process to not block GUI). When it detects a phrase, it hands it to Ultron's main logic as if it were a typed query. In the current logs, we saw `ultron_voice_input.txt` being used as a way to capture voice commands; you might continue with that approach (monitoring a text file) or directly integrate into the GUI (e.g., a "Start Listening" button that uses the microphone). - **Voice Output (TTS)**: Ultron could speak responses. Designing for this, you might leave placeholders where after generating a text response, you could call a TTS engine (offline example: `pyttsx3` library, which works without internet; online example: Google TTS or Azure). For now, maybe just ensure the code is structured so that adding a TTS call is straightforward (perhaps in the place where you update the GUI with the response, also call a `speak(answer)` function if voice is enabled).

GUI hooks for voice: You could add a microphone icon button in the GUI for "start/stop listening", whose callback would interface with the `ultron_voice` module. Or if always listening, show a status

like “ Listening…” when active. These are design choices for later, but by separating concerns (logic vs UI vs voice I/O), you make it easier to implement later. For now, maybe include a note in the code like:

```
# TODO: Integrate voice recognition here.
```

and ensure the thread architecture can accommodate continuous listening (which might produce input asynchronously).

Finalizing and Populating GitHub Repository

With all components developed, double-check **completeness, portability, and usability**: - **Completeness**: Does Ultron cover the basic functionality of responding to inputs using the AI models? Ensure nothing from the old system is lost (e.g., if it used to watch a folder for text or voice input, either keep that or replace with an interactive GUI input). Provide instructions for any new usage pattern. - **Portability**: Test on a different PC (without your dev environment). Does it run off the USB on a fresh Windows install? If it complains about missing DLLs (like VC++ redistributables), you might need to include those. PyInstaller usually bundles the Python runtime, but not always VC++ runtime – if using things like PyQt or others, ensure the needed dlls are present. - **Usability**: Is the GUI intuitive enough? Perhaps get a friend or colleague to try it and give feedback. Simple text and buttons should be fine for now.

Now, publish the source code to the GitHub repo: 1. Push all Python source files (`ultron_main.py`, etc.), the `requirements.txt`, and the README/documentation. 2. Include the PyInstaller spec or notes about building in the README (for developers). 3. Do **not** push the actual model file (due to size) or any secret keys. Instead, in README, guide the user how to obtain the model (e.g., “Download `orca-mini-3b.ggmlv3.q4_0.bin` from GPT4All’s model zoo and place it in the `models/` folder next to the exe.”). 4. Provide a simple **installation/use guide**: - Prerequisites (e.g., “Install Python 3.x and PyInstaller if building from source”). - How to run from source (for developers) vs how to use the packaged EXE (for end users). - Mention the offline/online behavior and config file usage. - Mention the Google Drive sync feature (and how to change the path if someone has a different setup).

Finally, after populating the repo, the GitHub can serve as the central place for code maintenance. Since the user specifically said “populate with full source and build script”, make sure the build process is well documented (maybe in a section of README titled "Building the Executable").

With all these steps, Ultron will be compiled into a portable .exe that meets all the requirements: it will run from a USB stick on Windows without installation, use relative paths so it’s self-contained, prefer offline operation via a local model (with OpenAI as backup), present a GUI for user feedback, sync data to Google Drive when possible, and be structured to allow future voice capabilities to be added.

References:

- Using `sys.frozen` and `sys.executable` for portable file paths [1].
- GPT4All local model usage (custom model path and offline mode) [4] [5].
- PyInstaller one-file Windows executable example [6].

---

1 executable - Determining application path in a Python EXE generated by pyInstaller - Stack Overflow
https://stackoverflow.com/questions/404744/determining-application-path-in-a-python-exe-generated-by-pyinstaller

2 3 GPT4All Python SDK - GPT4All
https://docs.gpt4all.io/gpt4all_python/home.html

4 5 Python Bindings · nomic-ai/gpt4all Wiki · GitHub
https://github.com/nomic-ai/gpt4all/wiki/Python-Bindings

6 python - How to use pyinstaller? - Stack Overflow
https://stackoverflow.com/questions/34453458/how-to-use-pyinstaller