ChatGPT

Developing an AI-Powered Radial Menu Assistant for Windows

Overview and Concept

The goal is to create a **native Windows application** that acts as an AI-powered assistant, triggered via the middle mouse button. When the user middle-clicks, a **radial menu UI** (an "emote wheel"-style menu) appears around the cursor, offering context-aware AI-generated options. These options (prompts or actions) are tailored to the active window or on-screen content. The tool will incorporate voice input and screenshot analysis to suggest the *next best prompts* for the user, helping them progress in building their application or performing their current task.

This concept builds on ideas from existing tools. For example, the *Pie Menu* app on macOS allows a radial menu around the cursor with options customized per active application [1]. On Windows, open-source projects like **AutoHotPie** have implemented global radial menus for macros [2]. Moreover, recent OS features show the potential of on-screen AI: Microsoft's *Click to Do* (Windows 11) analyzes screen content to offer actions (like copying text or searching the web) [3], and Apple's *Visual Intelligence* in iOS 26 even lets users send a screenshot to ChatGPT for analysis [4]. Leveraging these ideas with the latest AI models (OpenAI's GPT-4 with vision, Whisper for speech) will allow us to create a cutting-edge assistant.

Core Features and Requirements

- **Radial Menu Triggered by Middle Click:** Replacing or augmenting the normal middle-mouse behavior. When pressed, it should **open a GUI radial wheel** at the cursor. The user can then select an option by moving the mouse toward one of the menu slices and clicking or releasing, similar to game emote wheels.
- **AI-Generated Contextual Options:** The menu options are not static; they are generated by an AI based on the current context (the active window's content). For example, if the user is coding in an IDE, the options might be prompts to ask for debugging help or code suggestions. This requires analyzing the screen (or active window) to understand what the user is doing.
- **Screenshot Analysis:** The app should **capture a screenshot** of the current window or screen when triggered. Then an AI model will process this to determine the context and come up with the *optimal next prompts* for the user. By 2025, OpenAI's GPT-4 model can accept image input (as seen with ChatGPT's vision features) [4], so we can utilize that for analysis. If image input is unavailable, the app can fallback to extracting text (OCR) from the screenshot and sending that to the AI.
- **Voice Input Integration:** The radial menu will include a **voice record button**. Pressing it lets the user speak a query or command, which the app will record and send to the AI (using OpenAI's Whisper or a similar speech-to-text service) for transcription. The transcribed text can then be sent to ChatGPT to handle the query. (OpenAI's Whisper API can accurately transcribe audio to text [5].)
- **System-Wide Functionality:** The app runs in the background and works across all applications (system-wide). It must hook into global input so that the middle-click is recognized anywhere (browser, IDE, design tool, etc.), and it should overlay the radial menu on top of other windows.

- **AI-Powered Prompt Suggestions:** The core purpose is to help users figure out *what to ask next*. The AI will generate a list of suggested prompts/questions that the user might want to ask ChatGPT, given what's on their screen. These suggestions will be displayed as the menu options. Selecting one will copy it to clipboard or directly send it to ChatGPT (for example, via the ChatGPT API or by pasting into an active chat interface).
- **Open Source and Extensible:** The project will be open-sourced on GitHub (the user's repository is provided). It should be free for everyone, which means it will likely require users to provide their own OpenAI API key to use the AI features (to avoid running costs for the developer). We will choose a permissive license (like MIT) so the community can contribute. The design should be modular to allow adding more languages later and integrating other models or services.

Technical Plan and Step-by-Step Development Guide

Developing this application involves multiple components: global input hooking, a GUI (radial menu overlay), screenshot capture and processing, integration with OpenAI's APIs (for vision, language, and voice), and glue to tie it all together. We will use **ChatGPT (and Codex)** as a coding assistant throughout development, prompting it for help with specific tasks. Below is a step-by-step plan, including example prompts to use with ChatGPT for each stage, and directions on implementing the features using the latest 2025 technologies.

Step 1: Research and Ideation

Before coding, gather knowledge on similar projects and required techniques: - **Research Similar Tools:** Look at projects like *AutoHotPie* (radial menus via AutoHotkey) and *Pie Menu for Mac*. AutoHotPie's approach uses an Electron GUI with AutoHotkey scripts for input hooking [2]. This informs us that we need global hooks and a way to draw a circular menu overlay. Pie Menu (macOS) confirms the usefulness of app-specific menus [1], which we aim to automate with AI. We also note Microsoft's and Apple's recent AI features for screen content – proving the concept's feasibility [4] [3]. - **Decide Tech Stack:** Given the need for low-level input capture and a rich UI, a good choice is **C# with .NET** (latest version, .NET 8 in 2025) for a native Windows app. .NET provides access to Win32 APIs for hooks and a robust UI framework (WPF or WinUI). WPF is a mature framework that works well for desktop apps and allows custom controls and overlays. There are existing WPF controls for radial menus we can reuse [6]. Alternatively, WinUI 3 (Windows App SDK) could be used for a modern UI style; however, WPF's ecosystem and ease of global hooks make it a safe choice. - **AI Services:** We will use OpenAI's APIs. **ChatGPT (GPT-4)** will generate the prompt suggestions. By 2025, GPT-4's image analysis (vision) capability is accessible, enabling us to send a screenshot and get relevant output [4]. For voice, we'll use **OpenAI Whisper** for speech-to-text, since it's highly accurate and accessible via API [5]. (OpenAI's newer text-to-speech can be used if we want the assistant to talk back, but that's optional.) - **Global Hooking Strategy:** We need to capture the middle mouse click globally. Windows offers low-level hooks (`WH_MOUSE_LL` and `WH_KEYBOARD_LL`) for global input monitoring [7]. We can either use P/Invoke to call Win32 API `SetWindowsHookEx` or use an existing library. There's an open-source library **Gma.System.MouseKeyHook** (by George Mamaladze) that simplifies global hooks in .NET [7]. Another modern option is **SharpHook** for .NET (cross-platform global hooks) [8]. Using a library can save time and ensure reliability. We must ensure that when our hook triggers the radial menu, it can **suppress** the normal behavior of the middle-click (e.g., prevent auto-scroll or link opening in browsers). - **UI Overlay:** The radial menu should appear as an overlay on top of other windows. This likely means creating a borderless WPF window that is transparent and always on top (topmost), or using a full-screen transparent overlay that draws the menu at the cursor position. We'll ensure the window does not

steal focus (so it behaves like a context menu). WPF can handle transparent windows with `AllowsTransparency=true` and `WindowStyle=None`. We might set the overlay window to click-through except for the menu itself (so it doesn't block other interactions). - **OpenAI Integration:** Plan how the app will communicate with OpenAI. For GPT-4, use the REST API endpoints (ChatCompletion API). For image input, use the multi-modal endpoint (if available) or an alternate approach: possibly splitting into uploading image to a host or using the new function calling if OpenAI provides an image-to-text function. For Whisper, use OpenAI's `/v1/audio/transcriptions` endpoint. We'll need to manage API keys and consider rate limits. - **Example Prompt (Research):** To have ChatGPT help in this planning phase, ask something like:

Prompt: *"I'm designing a Windows app that opens a radial menu on middle-click. It will capture the screen and use GPT-4 to suggest next-step prompts for the user. How can I globally hook the middle mouse button in C#, and what frameworks or libraries might help me draw a radial menu overlay? Are there any open-source examples?"*

ChatGPT can then explain about using `SetWindowsHookEx` for low-level mouse hooks, mention libraries like globalmousekeyhook or SharpHook, and suggest UI frameworks (WPF with custom drawing or ready-made controls). It might even find the RadialMenu.WPF library or advise on how to draw circular menus in WPF.

Step 2: Project Setup

With a clear plan, set up your development environment and project structure: - **Initialize a Repository:** Create a new repository (since this will be open source, a GitHub repo has been specified). Set up a README to document the project goals. - **Create the Solution:** Using Visual Studio or the .NET CLI, create a new C# project. A WPF Application project (.NET 8) is suitable. This gives us an App.xaml (for global resources) and a MainWindow by default. - **Add Necessary Packages:** If using the **MouseKeyHook** library for global hooks, install it via NuGet (e.g., `PM> Install-Package Gma.System.MouseKeyHook`). If using **RadialMenu.WPF** control (an open-source control library [6]), add that via NuGet as well (`RadialMenu.WPF` package). Also add packages for OpenAI API integration. OpenAI provides a .NET SDK, or we can use HttpClient to call the REST endpoints directly. If an official OpenAI .NET SDK exists (by 2025, there might be one), include that for convenience. - **Example Prompt (Setup):**

Prompt: *"Help me set up a new WPF project in C# that will use a global mouse hook and a radial menu control. Which NuGet packages should I install for global keyboard/mouse hooks and for a radial menu UI? Provide the CLI or Visual Studio steps."*

ChatGPT might respond recommending a library like MouseKeyHook [7] for hooks and mention RadialMenu.WPF or others for the UI, giving installation commands. It could also mention ensuring app manifest settings if needed (for example, requiring UIAccess if we wanted to capture inputs on secure desktop, though not needed here since it's just general use).

- **Set Up OpenAI API Access:** You'll need an API key from OpenAI. Plan to load this from a config file or environment variable, rather than hard-coding, for security. In the code, store it in a secure manner (not in the repo). We might prompt ChatGPT for how to securely load API keys in a .NET app (like using user secrets or reading from a config).

Step 3: Implement Global Middle-Mouse Hook

Enable the application to detect when the user presses the middle mouse button anywhere: - **Using the Hook Library:** If using `Gma.System.MouseKeyHook`, you can set up an event for mouse clicks. For example, the library allows subscribing to `Hook.GlobalEvents().MouseDownExt` which gives global mouse events, or using `IKeyboardMouseEvents`. Specifically, listen for `MouseButtons.Middle`. Once detected, **show the radial menu** and suppress the event's default effect. The library's ExtendedMouseEventArgs might let you mark the event as handled (or if not, we might need the lower-level approach to actually block it). - **Manual Hook (Alternate):** If we choose to manually use `SetWindowsHookEx(WH_MOUSE_LL)`, we'll write a LowLevelMouseProc callback in C# via P/Invoke. In that callback, filter for WM_MBUTTONDOWN. To prevent the event from propagating, the hook callback can return a non-zero value (instead of calling `CallNextHookEx`) when the middle button is pressed [7]. This will **block the event** from reaching other applications while our menu is open. (Be careful to unhook properly on app exit.) - **Testing the Hook:** For now, just have the handler do something simple, like logging or showing a MessageBox, to confirm it triggers on middle-click globally. - **Example Prompt (Hook Code):**

Prompt: *"Show me a C# code example of using SetWindowsHookEx to set a low-level mouse hook (WH_MOUSE_LL) and detect middle mouse button clicks globally. I want to prevent the default action when middle-click is pressed."*

ChatGPT should provide a code snippet with PInvoke declarations for `SetWindowsHookEx`, `LowLevelMouseProc`, etc., and logic to intercept the event. It may also mention the need for `[DllImport("user32.dll")]` and possibly running the message loop (though in WPF, the message loop is hidden in Application.Run). Since WPF is an STA thread for UI, we might need to ensure the hook is set on a background thread or the main thread is pumping messages – typically, setting the hook on the main thread should be fine as WPF's dispatcher loop will keep it alive.

- **Ensure Thread Safety:** If the hook callback runs on a separate thread, ensure any UI calls (like opening the menu) are marshalled to the UI thread (Dispatcher.Invoke in WPF). We may prompt ChatGPT about updating WPF UI from a hook callback if needed.

Step 4: Designing the Radial Menu UI

Next, create the GUI for the radial menu: - **Using a Radial Menu Control:** To save time, consider using the open-source **RadialMenu.WPF** control [6] (by Julien Marcou). This control allows defining RadialMenu and RadialMenuItem elements in XAML and supports multiple slices and even nested submenus. We can style it as needed. The radial menu can be a UserControl or just XAML in a window that becomes visible on trigger. - **Basic Layout:** Design a RadialMenu with a central hub (maybe an icon or "⚙" for settings or " " for voice) and several outer slices for the prompt suggestions. Initially, you may hard-code a few dummy menu items to test the look and feel. For example, create 6 or 8 RadialMenuItem in XAML with placeholder text (like "Option 1", "Option 2"...). - **Overlay Window:** The RadialMenu will reside in a hidden Window that we show on middle-click. Create a WPF Window (let's call it `RadialMenuWindow`) with `WindowStyle=None`, `AllowsTransparency=True`, `Background=Transparent`, `Topmost=True`. In that window, place the RadialMenu control. We might bind the RadialMenu's center to the window's center or to the mouse position. - **Positioning at Cursor:** When showing the menu, set the window's Left and Top to be at (cursorX - windowWidth/2, cursorY - windowHeight/2) so that the menu center aligns with cursor. Alternatively, if the RadialMenu control itself allows specifying an origin, use that. - **Appearance:** Style the menu slices with icons or text clearly visible. We may use simple text labels for the suggestions (the prompt text might be

long, so perhaps show a short summary on the slice and full text on hover tooltip). - **Non-intrusive:** Make the window click-through except for the menu itself. This can be done by setting `IsHitTestVisible=false` on the transparent areas, or using Window's `AllowsTransparency`. We want only the menu items to be clickable. - **Example Prompt (UI Design):**

Prompt: *"Give me an example of a XAML layout for a radial menu in WPF with 6 items. Ideally using an existing RadialMenu control. Each menu slice should have text content. The window should be transparent and topmost."*

ChatGPT might provide a snippet using a RadialMenu control (if it knows about the NuGet) or show how to create a custom control with Path geometry for slices. It might output something like the usage shown in the RadialMenu.WPF docs, e.g. defining `<RadialMenu:RadialMenu IsOpen="True" Radius="150"> <RadialMenu:RadialMenuItem Content="Option1"/> ... </RadialMenu>`.

For example, a custom WPF RadialMenu control displays menu items around a center button. The image above shows a RadialMenu with several labeled slices ("New file", "Edit", "Save", etc.) arranged in a circle [9] [10]. We can adopt a similar UI approach: each slice in our menu will correspond to an AI-generated suggestion, and the user can select it by moving the mouse in that direction. Using an established radial menu library saves development time and ensures reliable rendering of the circular layout.

- **Central Button and Voice Input:** Design what the center of the radial menu does. It could be a "cancel/close" button, or possibly a voice-input trigger. For instance, the center could show a microphone icon – clicking it starts recording audio. Alternatively, we use one of the slices for voice input. We'll refine this once basic menu works.
- **Show/Hide Logic:** By default, keep the RadialMenuWindow hidden. When middle-click occurs (as handled in Step 3), gather AI suggestions (Step 6) and then show the window. When an option is clicked or the menu loses focus (user clicks elsewhere or presses ESC), hide the window and possibly re-enable normal mouse behavior.

Step 5: Capturing Screenshot of Active Window

Implement functionality to capture the screen or window for context: - **Determine Target (Window or Full Screen):** Capturing the active window is ideal to limit scope. We can use Windows API to get the **foreground window handle** (`GetForegroundWindow`) and its bounds (`GetWindowRect`). Then capture just that region. Alternatively, capture the entire desktop screen if we want all visible context (multi-monitor considerations apply). For initial simplicity, capturing the **primary screen** or the entire virtual screen is okay, then we can refine to just active app window. - **Capture Method:** In C#, one simple way is using `Graphics.CopyFromScreen` to copy the screen pixels into a Bitmap. Another way is P/Invoke GDI functions (BitBlt). Example using System.Drawing:

```
var bmp = new Bitmap(width, height, PixelFormat.Format24bppRgb);
using(var g = Graphics.FromImage(bmp)) {
    g.CopyFromScreen(captureX, captureY, 0, 0, new Size(width, height));
}
// bmp now has the screenshot
```

If capturing a specific window, adjust `captureX, captureY` to the window's top-left coordinates and width/height to window size. (Note: if the window is covered by others or minimized, CopyFromScreen won't capture it properly. To capture off-screen or covered windows, one would need DWM thumbnail or similar – out of scope for now since we assume user's active window is visible.) - **Prepare Image for AI:** The image can be kept in memory. The OpenAI API may require us to send it as a file or byte stream multipart form. We might save it to a temporary file (e.g., PNG or JPG) before sending, or use an in-memory stream. - **Example Prompt (Screenshot Code):**

Prompt: *"How can I capture a screenshot of the active window in C#? I have a window handle (IntPtr). Show an example of using BitBlt or Graphics.CopyFromScreen to get an image of that window."*

ChatGPT should provide a code example using `User32.GetWindowRect` and `GDI32.BitBlt` or the simpler CopyFromScreen method. It may mention using `PrintWindow` API for hidden windows, but for active window that's visible, CopyFromScreen suffices.

- **Active Window Title (Context):** Additionally, get the active window's title and process name (via `GetWindowText` and perhaps `GetModuleFileNameEx` for the process). This can be fed to the AI as well to give context like "User is currently in [VS Code - file name]" which might help the AI tailor suggestions. For example, if the title contains "Visual Studio" or "Chrome - StackOverflow", the AI can infer context.
- **Timing:** The screenshot and OCR/analysis will take a moment. We might capture as soon as middle-click is pressed, then show a loading indicator in the radial menu while the AI is formulating suggestions. Possibly the menu appears with placeholders (spinners) until the suggestions arrive.

Step 6: Integrating OpenAI API for Vision and Prompt Generation

Now the key part: using AI to get prompt suggestions from the screenshot. - **Using GPT-4 Vision:** The ideal route is to send the screenshot image to GPT-4 and ask: *"Given this screen, what are some helpful things the user might want to do or ask next?"*. OpenAI's documentation should be checked for how to send an image in the ChatCompletion API. Likely, one provides the image as an attachment in the API call (the API might accept a URL or a base64 image in 2025). If it's complicated, an alternative is using an **OCR + text prompt** approach: - Run OCR on the screenshot to get visible text. Microsoft has a Windows.Media.Ocr API for UWP, or we could use Tesseract OCR library. However, OCR only gives text, not the full context (images or UI state). Still, text like error messages, code, or UI labels can be very useful. - Identify the application (maybe from window title or recognizable UI elements). - Formulate a prompt for GPT-4: e.g. *"The user is working in [Application]. The screen contains the following text:* `...` *(include OCR text or descriptions). Suggest 5 next questions or actions the user might want to do to make progress on their task."* - GPT-4 (text-only) would then infer context from the text and respond with ideas. - **Prompt Engineering:** We want the AI's output to be a list of prompt suggestions. We should instruct it clearly. For example: *"You are an assistant to help users figure out next steps in their project. Look at the screen's content and propose a list of 3-5 insightful questions or commands the user could ask an AI to help them. Respond with a bullet point list of suggestions."* This system/user prompt can be relatively static, combined with the dynamic screenshot input. By giving GPT a role and an example if possible, we can get better results. - **API Call:** Implement a function that takes the screenshot (or OCR text) and calls the OpenAI API. Using the ChatCompletion API with a properly crafted message array (system message with role/instructions, user message with context). The model will return a response containing the suggestions. Parse the response (if it's in bullet form or numbered list). - **Error Handling:** If the AI response is too long or not in the expected format, we may need to parse or even instruct the model

to answer in a concise list. Also handle API errors or slow responses (maybe time out after a few seconds and show an error or retry). - **Example Prompt (OpenAI API integration):**

Prompt: *"Using OpenAI's C# API or HttpClient, how can I send an image to GPT-4 for analysis? If direct image input isn't available, what's a good way to use OCR text as context in a GPT-4 prompt? Provide a code example for calling the OpenAI ChatCompletion API with a user message and system message."*

ChatGPT might explain that as of 2025, the API supports images in the chat (perhaps by including it as a special message with `"image": "<base64 or file>"` or some new parameter). If not, it will suggest doing OCR and then sending a text. It will provide an example of constructing an HTTP request or using OpenAI SDK: setting `model="gpt-4-vision"` and including the image. Or using GPT-4 (without vision) and including text context.

- **Citation Note:** OpenAI announced multi-modal capabilities; for instance, Apple's integration shows uploading a screenshot to ChatGPT is possible [4], which implies by mid-2025 developers can utilize similar functionality.

- **Testing with a Sample:** You might test with a known scenario. For example, if the user's screen has a compiler error in code, feed that text to the prompt and see if GPT-4 suggests relevant prompts ("Ask how to fix the error...", "Suggest a different approach to implement X", etc.). Tweak the prompt instructions until the suggestions are relevant and actionable.

Step 7: Populating and Displaying AI Suggestions

Connect the AI output to the radial menu UI: - **Updating Menu Items:** Once the suggestions (as text strings) come back, create or update the RadialMenu items. If using a bound ItemsSource, update the collection. With RadialMenu.WPF, you might need to replace the Items collection entirely to refresh [11]. Ensure this happens on the UI thread (use Dispatcher.Invoke if the AI call was on a background thread). - **Menu Item Content:** The suggestions could be full sentences or questions. They might be long. For the radial button labels, use a shortened version if necessary (perhaps the first few words) and show the full prompt on hover tooltip. Alternatively, number them and show a number on the slice, with the full text in a legend or center when hovered. You can be creative – maybe when a slice is highlighted, the center of the wheel displays the full text. - **Show the Menu:** Now that we have content, display the RadialMenuWindow (set `Visibility=Visible` or call `Show()` if it was closed). The menu appears under the cursor. The user can then move and click on a slice. - **Selecting an Option:** When the user picks one of the suggestions, decide what happens: - Simplest: **Copy the prompt text to clipboard** and close the menu. The user can then paste it wherever (e.g., into ChatGPT's chat if they're using the web UI, or any text box). - Advanced: If we integrated the ChatGPT API fully, we could directly send this prompt to ChatGPT and perhaps display the answer. However, that requires building a chat UI or some way to present the response (maybe a popup or in the menu itself). To keep scope manageable, copying to clipboard or even auto-pasting into an open chat window (if we detect one) is okay. (Auto-paste might involve sending keystrokes – could be unreliable. Clipboard + instruct user to paste might be sufficient.) - **Visual Feedback:** After selection, provide feedback like a brief highlight or a notification ("Prompt copied!"). Then hide the radial menu until next time. - **Example Prompt (UI Update Code):**

Prompt: *"How can I update a WPF RadialMenu's items at runtime? Suppose I have a list of strings from an AI response. Show how to populate a RadialMenu control with these as RadialMenuItem and handle the click event to copy text to clipboard."*

ChatGPT might produce C# code: create `new RadialMenuItem { Content = "Suggestion 1" }` for each string, attach a Click handler that uses `Clipboard.SetText(suggestionText)`, and then assign them to the RadialMenu. It will likely remind to use `Dispatcher` if coming from another thread.

- **Clipboard Handling:** Use `System.Windows.Clipboard.SetText()`. On Windows, clipboard operations require STA thread (which WPF UI already is). Ensure the text is properly set. Test that it works by selecting an option and pasting in Notepad or ChatGPT.

Step 8: Voice Input Integration

Add the ability for the user to record a voice query and send it to ChatGPT: - **Recording Audio:** Use a library or API to record microphone input. One popular .NET library is **NAudio** for working with audio. Alternatively, by 2025 Windows might have improved Media APIs for audio capture in .NET. We can record a few seconds of audio from the default microphone when the user holds the voice button. - Implement a button (e.g., center of radial menu or a special slice). When pressed, start recording (and maybe indicate with an icon change or animation). When released, stop recording and save the audio to a file (WAV or MP3). - **Example Prompt (Recording):** *"Using C# and NAudio, how do I record audio from the microphone and save it to a WAV file? Provide a short example."* - ChatGPT might show how to use `WaveInEvent` to capture audio, accumulating buffers and writing to WaveFileWriter. - **Sending to Whisper API:** Take the recorded audio file and send it to OpenAI's Whisper API for transcription. The Whisper API (`POST /v1/audio/transcriptions`) expects an audio file and returns text. Use HttpClient to POST the file with the API key. The response will be JSON with the transcribed text. - **Handling Transcription:** Once we get the text, we can either: - Directly send that as a query to ChatGPT (and maybe read out or display the answer). - Or populate it as a suggestion in the radial menu? This doesn't quite fit the "suggestion" paradigm. More likely, voice input is a separate feature: basically a voice query to ChatGPT. Possibly the user might ask a question not covered by suggestions. - Implement a simple flow: after transcription, call the ChatGPT API with the transcribed text as a user query, get the answer, and display it in a popup or maybe read it aloud. - Reading aloud could use text-to-speech (Windows has `SpeechSynthesizer` or use Azure TTS). However, text display is simpler at first. - **UI for Voice Q&A:** We might add a basic chat dialog or just a message box with the answer. Or integrate it into the radial menu: for example, show the answer in the center of the wheel or as another overlay. Since the focus is on prompt suggestion, this might be an auxiliary feature. - **Example Prompt (Whisper API call):**

Prompt: *"How can I send an audio file to OpenAI's Whisper API in C# to get a transcription? Show an example using HttpClient with multipart/form-data."*

ChatGPT should outline making an HTTP request with the file content and the required parameters (`model: "whisper-1"` etc.), and reading the response.

- **Connecting it:** Put a microphone icon on the radial menu (say in the center). When clicked, begin recording (perhaps toggle the icon to a stop button or a blinking indicator). After releasing/stop, do the Whisper call, get text, then call ChatGPT with that text. Finally, show the answer. For a better user

experience, this might be done outside the radial menu context (maybe minimize the menu while processing and then show a result window).

- **Voice to Improve Prompts:** Another interpretation is the voice input could be used to **ask the assistant to refine the suggestions**. However, likely it's meant for direct questions to ChatGPT when needed.

Step 9: Context Awareness and Refinements

Improve how well the suggestions align with the user's context: - **App-Specific Prompts:** If possible, tailor the AI prompt with the active application name or type. For instance, if the active window's process name is `devenv.exe` (Visual Studio) or the title contains "Visual Studio Code", include a hint: "The user is coding in [VS Code]" so that GPT knows to focus on programming-related help. If it's a browser on Stack Overflow, GPT might suggest searching strategies or explaining an answer, etc. - **Multi-language Support (Future):** Initially, we focus on English (as specified). But design the prompt generation such that it could be parameterized for language. For example, if later adding Spanish, we could instruct GPT to respond in Spanish. The UI should be able to display Unicode characters for other languages. We might not implement now, but keep in mind (e.g., avoid hard-coding English text in UI elements that might need translation). - **Performance Considerations:** Calling GPT-4 with an image can be slower (maybe a few seconds). Whisper for audio also takes a few seconds depending on length. We should provide feedback to the user that suggestions are loading. Perhaps a simple "Loading AI suggestions..." overlay or spinner on the radial menu while we await the response. This can be done by initially showing the menu with some dummy slices or a spinner icon, and updating once ready. - **Rate Limiting & API Errors:** Implement basic handling if the OpenAI API fails (network error, rate limit, etc.). In such case, perhaps fall back to a generic set of prompts (like "Ask for help", "Explain this error") so the menu isn't empty, or show an error message. - **Privacy:** Since screenshots could contain sensitive information, make sure to inform users (in README or UI) that images and text are being sent to OpenAI. Encourage them to use it on non-sensitive content, or implement an option to exclude certain apps (for example, don't capture screen if the window title matches certain keywords like "Password" or "Bank"). As an open source project, users can inspect what's sent. Also, no data is kept by the app itself – it's either on the user's machine or sent to OpenAI's servers (subject to their privacy policy). - **Example Prompt (Refining AI Prompt):**

Prompt: *"I want the AI suggestions to be tailored to the app. If the active window title is 'Photoshop', suggestions should be design-related; if it's code, then coding help. How can I use the window title or process name in the GPT prompt to improve relevance?"*

ChatGPT may suggest building a few rules or simply including the app name in the prompt as context. It might also suggest using specialized instructions like "If application is X, do Y" – though we will rely on GPT's general ability mostly.

Step 10: Testing the Application

Thoroughly test each component: - **Hook Reliability:** Verify that the middle-click does trigger the menu in various applications (browser, text editor, etc.), and that the default middle-click action is indeed suppressed. If you find situations where middle-click still goes through (e.g., some games or admin windows), you might need to run the app with higher privileges or adjust the hooking method. - **UI Behavior:** Test that the radial menu appears under different screen resolutions and DPIs correctly at the cursor. Ensure it disappears

when an option is chosen or on cancel (you might allow right-click or ESC to cancel out of the menu without choosing). - **AI Suggestions Quality:** Try different contexts: - Open a code editor with some code and error message visible, see if suggestions revolve around debugging or explaining the error. - Open a documentation webpage, see if it suggests summarizing or asking questions about it. - If suggestions are not useful, adjust the system prompt given to GPT, or provide more of the screen context (maybe more text). - **Voice Queries:** Test the voice feature by asking a variety of questions. Check the transcription accuracy and that the answer from ChatGPT is received and displayed. If it's slow, consider adding a loading indicator or audio "ding" when ready. - **Resource Usage:** Ensure the app isn't using too much CPU or memory when idle. Hooks and a hidden window are lightweight. The heavy parts are the AI calls. Perhaps implement a cooldown or ensure we don't accidentally send multiple requests if the user spams the middle-click. - **Edge Cases:** What if the user triggers it rapidly multiple times? Possibly debounce the input. Also, if no internet connection, handle that gracefully (show a message "AI services unavailable").

Step 11: Deployment and Open Source Release

Prepare the app for public use: - **Open Source Prep:** Double-check that no sensitive info (like API keys) are in the repo. Provide instructions in the README for users to get their own OpenAI API key and configure the app (maybe via an .config file or the UI prompt on first run). - **Choose a License:** The user wants it open for everyone, so an MIT or Apache 2.0 license is appropriate (MIT is straightforward and commonly used for GitHub projects [12]). Include a LICENSE file. - **Documentation:** Write a detailed README with screenshots of the app in action, and usage instructions. Explain how to install (we might provide a compiled installer or exe in Releases). Mention the features: global radial menu, AI prompt suggestions, voice input. Also include a disclaimer about OpenAI costs (users will need an API key which may incur charges for usage). - **Build and Package:** Use a tool like MSIX or simply distribute as an EXE installer using something like Inno Setup. Ensure the app can run on Windows 10 and 11. If using .NET 8, you might publish self-contained so users don't need the runtime installed. Testing the installer on a fresh machine is a good idea. - **Continuous Improvement:** Encourage feedback. Since it's open source, others might contribute improvements (e.g. adding support for multiple languages, custom prompt templates per application, etc.). We can also integrate updates as OpenAI releases new models (the plan should be flexible to allow swapping in new model endpoints, like GPT-5 when available). - **Community and Accessibility:** Ensure the app is usable by different users. Possibly add a setting to change the hotkey if middle-click is not ideal for someone (maybe allow a keyboard shortcut as alternative). Also consider accessibility: for example, does the radial menu need keyboard navigation for those who can't use a mouse? It could be a future enhancement.

- **Example Prompt (README help):**

Prompt: *"Help me write a clear README for my open-source project 'Special Guide'. It's a Windows AI assistant that shows a radial menu of AI suggestions when you middle-click. Include setup steps, usage, and an example."*

ChatGPT can draft a nice README structure which we can refine and fill with specifics. It might also remind to add images or GIF demonstrating usage.

Conclusion

By following this step-by-step plan and utilizing ChatGPT's coding assistance at each stage, we can efficiently build the **"Special Guide"** Windows application. We leverage the best of 2025's technology: a modern .NET 8 WPF app for native integration, OpenAI's GPT-4 (with vision) and Whisper for powerful AI

capabilities, and proven libraries for UI and hooks. Each component — global input hooking, radial menu UI, screenshot analysis, prompt generation, and voice input — has been researched with up-to-date resources and integrated with best practices. The result will be a **free, open-source AI helper** that pops up on demand, helping users ask the right questions and move forward in their development or workflow, effectively acting as a context-aware copilot for any application.

Throughout development, **ChatGPT (Codex)** serves as our pair programmer, providing code snippets, debugging help, and architectural advice. By iteratively prompting ChatGPT for specific tasks (as illustrated in the examples), a solo developer can accelerate the implementation of even a complex multimodal application like this. The end product aims to empower everyone with an intelligent assistant at their fingertips — one that not only answers questions, but proactively suggests what to ask next, enhancing productivity and learning for all users.

Sources:

- Pie Menu concept on macOS (radial menu customized per app) [1]
- AutoHotPie – open source radial menus for Windows (using AutoHotkey + Electron) [2]
- Microsoft *Click to Do* – analyzes screen content for actions [3]
- Apple's Visual Intelligence – can send screenshots to ChatGPT for analysis [4]
- Global hook in .NET using WH_MOUSE_LL (CodeProject article) [7]
- RadialMenu.WPF – ready-made WPF control for radial menus [6]
- GPT-4 Vision and Whisper capabilities in 2025 (PyGPT assistant features) [5]

---

[1] Show HN: Pie Menu – a radial menu for macOS | Hacker News
https://news.ycombinator.com/item?id=41160268

[2] [12] GitHub - dumbeau/AutoHotPie: Radial menus in Windows, aka PIE MENYOOS!
https://github.com/dumbeau/AutoHotPie

[3] Click to Do: do more with what's on your screen - Microsoft Support
https://support.microsoft.com/en-us/windows/click-to-do-do-more-with-what-s-on-your-screen-6848b7d5-7fb0-4c43-b08a-443d6d3f5955

[4] Apple brings Apple Intelligence to the iPhone screen at WWDC 2025 | TechCrunch
https://techcrunch.com/2025/06/09/apple-brings-apple-intelligence-to-the-iphone-screen-at-wwdc-2025/

[5] PyGPT Desktop AI Assistant: o1, o3, GPT-4, GPT-4 Vision, ChatGPT, Gemini, Claude, Grok, DeepSeek, Perplexity, Ollama
https://pygpt.net/

[6] RadialMenu.WPF 1.3.0 - NuGet
https://www.nuget.org/packages/RadialMenu.WPF/

[7] Processing Global Mouse and Keyboard Hooks in C# - CodeProject
https://www.codeproject.com/Articles/7294/Processing-Global-Mouse-and-Keyboard-Hooks-in-C

[8] A cross-platform global keyboard and mouse hook for .NET : r/csharp
https://www.reddit.com/r/csharp/comments/qqd4xy/sharphook_a_crossplatform_global_keyboard_and/

9 10 11 GitHub - Julien-Marcou/RadialMenu: A WPF RadialMenu
https://github.com/Julien-Marcou/RadialMenu