– A Voice and Text Command Assistant for Windows

– By Somnath Das

TABLE OF CONTENTS

1 Introduction

1.1 Background

In recent years, voice-activated and text-based command systems have become increasingly popular due to their efficiency, accessibility, and user-friendly nature. From smart assistants like Alexa and Siri to command-line tools and automation scripts, there has been a continuous shift toward interactive and intelligent systems.

This project, titled "CmdMe – A Voice and Text Command Assistant for Windows", was developed as a desktop application for enhancing user interaction with the operating system. It merges the functionality of both voice recognition and traditional text input to execute common commands like opening applications, checking the time/date, or launching websites, all while offering a visually appealing graphical user interface (GUI).

1.2 Purpose of the Project

The main objective of CmdMe is to create a simple, lightweight, and user-friendly assistant that responds to both voice and text commands, specifically tailored for the Windows operating system.

The purpose includes:

Making routine tasks quicker and hands-free for users.

Helping beginners interact with the system without knowing command-line tools.

Demonstrating integration of speech recognition, text-to-speech, and GUI programming using Python.

It also serves as a practical application of the knowledge acquired throughout the Bachelor of Computer Applications (BCA) course.

1.3 Scope

The scope of this project is primarily focused on Windows systems and includes:

Voice-controlled operations like launching apps, getting the current time/date, or visiting websites.

Text-based input for performing the same actions as voice commands.

Dynamic chat-like interaction through a customized GUI using the CustomTkinter library.

Adjustable settings for appearance, voice rate, and voice gender to personalize the user experience.

However, this assistant does not support complex natural language understanding or external API integrations, and it requires installed apps like Chrome, Spotify, etc., to function as expected.

1.4 Features Summary

The key features of CmdMe are:

🎤 Voice and Text Input: Accepts commands both via microphone and keyboard.

🧠 Command Recognition: Understands a defined set of commands (like open Notepad, Chrome, etc.).

🕒 Time/Date Reporting: Responds with current time and date.

🌐 Website Launching: Opens `.com` websites mentioned in commands.

🖥️ System Commands: Supports shutdown and restart.

🎨 Customizable UI: Light/Dark/System themes, voice settings, and user preferences.

🗨️ Chat-style Interaction: Assistant replies displayed in a scrollable chat interface.

🔄 Reset and Clear: Easy reset of settings and chat logs.

This blend of functionality, visual appeal, and accessibility makes CmdMe a compact and powerful personal assistant for desktop users.

2 System Analysis

2.1 Existing System

In the current computing environment, users typically interact with their operating system through graphical user interfaces (GUI) or command-line tools. While these methods are effective, they can become repetitive or inconvenient, especially for non-technical users or those with accessibility needs.

Existing systems like Cortana, Google Assistant, or Alexa offer voice assistance but often require high system resources or cloud integration. Most command-line utilities, on the other hand, require memorization and offer no visual or voice feedback, making them less accessible to beginners.

There is a lack of lightweight, offline, customizable assistants that combine voice, text, and GUI on desktop platforms without heavy dependencies.

2.2 Proposed System

The proposed system, CmdMe, is a Windows-based desktop assistant that accepts both voice and text input from the user to execute simple commands such as:

Opening frequently used applications.

Launching websites directly.

Fetching and speaking out system time/date.

Executing basic system commands like shutdown/restart.

It uses Python libraries such as:

`speech_recognition` for voice input,
`pyttsx3` for text-to-speech,
and `os` for system command execution,

`CustomTkinter` for modern GUI.

Unlike existing systems, CmdMe is entirely offline, lightweight, and fully customizable by the user. It focuses on providing a clean user interface, ease of use, and interactive feedback without relying on external APIs or services.

2.3 Feasibility Study

Aspect	Feasibility
Technical	✅ Feasible – Developed using widely available Python libraries and runs on standard Windows systems.
Operational	✅ Feasible – Easy to use; GUI-based interaction makes it accessible to non-programmers.
Economic	✅ Feasible – No cost for development tools or libraries; all dependencies are open-source.
Legal	✅ Feasible – Project does not violate any licensing; open-source modules used.

Conclusion: The project is feasible in all aspects and implementable within the academic timeframe.

Hardware Requirements:

Component	Minimum Requirement
Processor	Intel i3 or equivalent
RAM	4 GB
Storage	250 MB (for code and dependencies)
Microphone	Required for voice commands
Speaker	Required for speech output
Operating System	Windows 10 or above

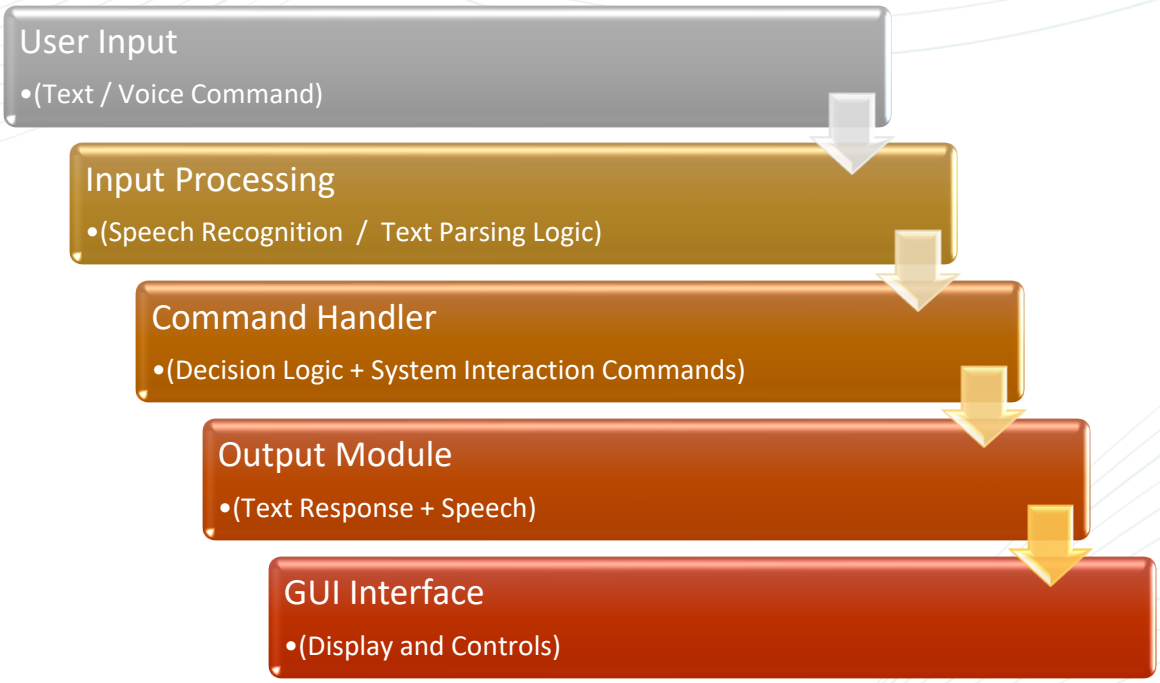
Software Requirements to build the app:

Software	Description
`Python (v3.8 or later)`	Main development language
`pip`	Python package installer
`CustomTkinter`	For GUI
`pyttsx3`	For text-to-speech
`SpeechRecognition`	For voice input
`Pyaudio`	For accessing microphone
`Subprocess, os, etc.`	For system-level interaction
`VS Code / Any IDE`	For development

3 System Design

3.1 Architecture Diagram

The architecture of CmdMe is designed in a modular and layered manner to ensure separation of concerns, maintainability, and scalability. The system comprises four primary components: Voice Command Module, Text Command Module, Command Handler, and Graphical User Interface (GUI).



3.2 Module Description

3.2.1 Voice Command Module

This module listens to the user's voice input through the microphone, processes it using the `SpeechRecognition` library, and converts it into text for further processing. The assistant uses **`pyttsx3`** to provide voice feedback.

Libraries Used: `SpeechRecognition, pyttsx3`

Error Handling: Unknown speech, network errors, or microphone issues are handled gracefully with proper error messages.

3.2.2 Text Command Module

This module allows the user to manually type commands into the GUI. It works similarly to the voice module but skips the recognition step.

Input Source: Text field in GUI

Output: Visual response in the chat display and optional voice feedback

3.2.3 GUI Interface

Built using `CustomTkinter`, this module provides a modern, theme-aware GUI. It includes:

Input field for typed commands

Buttons for "Send", "Speak", "Clear Chat", and Settings

Scrolled chat area showing the conversation

Sidebar for adjusting appearance and voice settings

3.2.4 Command Handler

The central logic of the assistant. It interprets the user's command and routes it to appropriate handlers:

App launch `(subprocess)`

System commands `(os.system)`

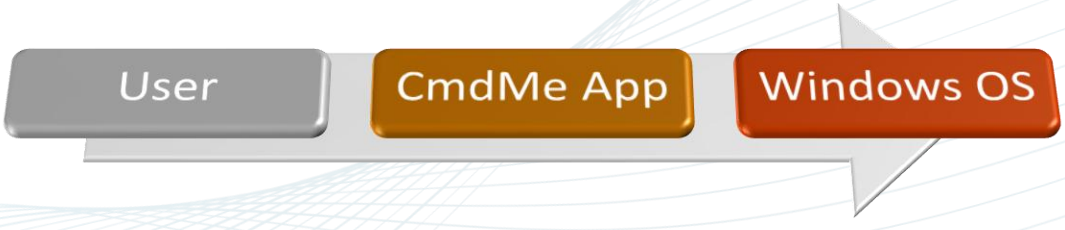
Time/date fetch `(datetime)`

Website open `(webbrowser)`

Each handler returns a response, which is then displayed and spoken to the user.

3.3 Data Flow Diagram (DFD)

Level 0 (Context Level)



Level 1 (Main Functional Breakdown)

3.4 UI Mockups

As the interface has already been implemented using `CustomTkinter`, the actual GUI serves as a live mockup. Key components of the GUI:

Main Chat Display Area – Shows assistant responses and user commands.

Input Box – Allows users to type commands.

Speak Button – Activates the microphone for voice input.

Sidebar Settings – Includes appearance mode (light/dark), voice rate, and gender settings.

Clear & Reset – To clear the chat and reset settings.

These design choices ensure the app is easy to use, visually clean, and accessible to both technical and non-technical users.

4 Implementation

4.1 Technologies Used

The implementation of **CmdMe** combines several Python libraries and frameworks to deliver a responsive, user-friendly, and functional desktop application. Below are the key technologies and their purposes:

Technology / Library	Purpose
`Python 3.x`	Core programming language
`customtkinter`	GUI framework for modern-looking widgets and theme support
`speech_recognition`	To capture and recognize voice commands from the user
`pyttsx3`	Text-to-speech engine for vocalizing assistant responses
`webbrowser`	To open URLs or websites directly from commands
`subprocess`	To launch desktop applications like Notepad, Calculator, etc.
`datetime`	For displaying current time and date
`threading`	To handle voice recognition without freezing the UI
`os`	To execute system-level commands (shutdown, restart, etc.)

These libraries were chosen for their simplicity, active community support, and integration capabilities with desktop-based systems.

4.2 Code Explanation (Key Modules)

Below is a breakdown of the main modules and their functionality:

a) Voice Command Module

- Captures audio using `speech_recognition.Recognizer.`
- Converts it to text using `recognize_google().`
- Handles errors like unknown speech or service failure.
- Example:

```
with sr.Microphone() as source:
    audio = recognizer.listen(source)
    command = recognizer.recognize_google(audio)
```

b) Text Command Module

- Accepts user input from the GUI entry field.
- Directly processes text without any conversion.
- Triggers on pressing "Enter" or clicking "Send".

c) Command Handler

- This is the central brain of the application.
- Matches user input with keywords like "open notepad", "what is the time", etc.
- Routes to:
  - App launch handler `(subprocess.Popen)`
  - Website handler `(webbrowser.open)`
  - System control `(os.system)`
  - Date/time response `(datetime.now())`

d) GUI (CustomTkinter)

- Designed with a sidebar (for settings) and a main panel (for chat and controls).
- Custom theme-aware buttons, sliders, and labels.
- Entry box and buttons like "Speak", "Send", and "Clear Chat".

e) Voice Engine Settings

- Adjusts voice rate and gender.
- Controlled by sliders and dropdowns in the GUI.

```
engine.setProperty('rate', rate)
engine.setProperty('voice', voices[0].id)  # male/female
```

f) Threading for Voice Input

- Voice listening runs on a separate thread.
- Ensures GUI stays responsive during long speech input.

```
threading.Thread(target=listen, daemon=True).start()
```

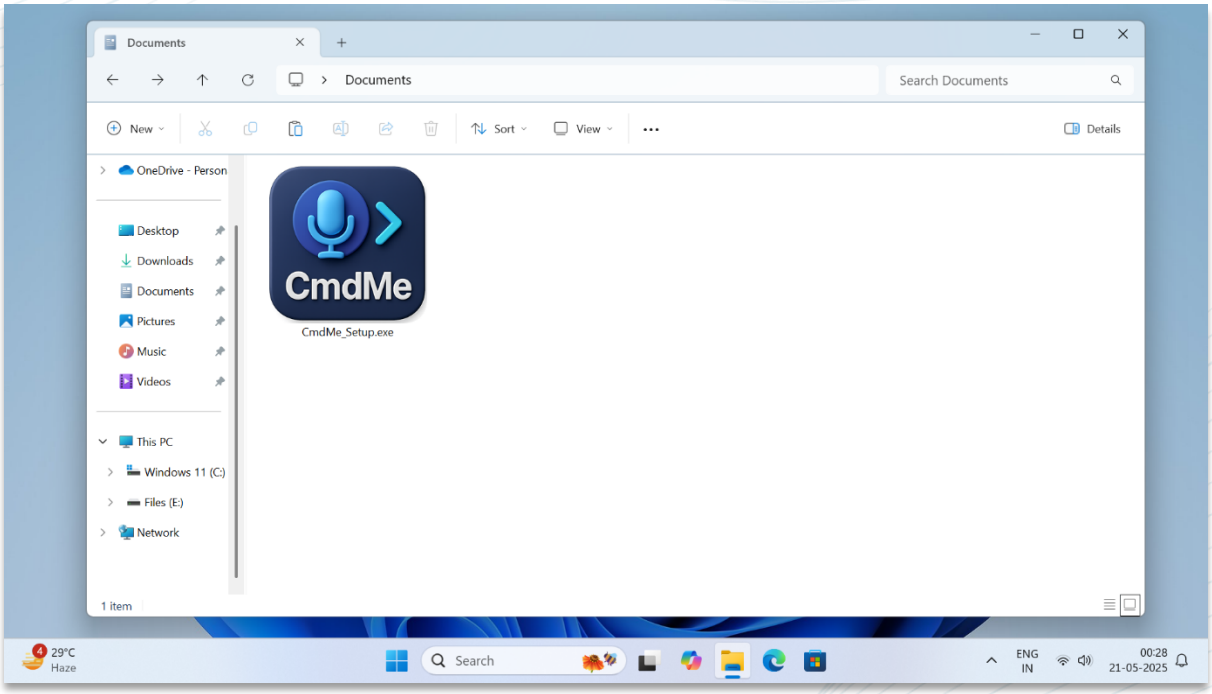
Each module is clearly separated in the codebase for ease of debugging, maintenance, and future upgrades. The modularity also allows integrating new features like file search or email support with minimal interference in the existing logic.

5 Installation Guide

This section describes how to install the application **CmdMe** using the setup executable created with *Inno Setup*. The setup wizard guides the user through the necessary installation steps.

5.1 Setup File

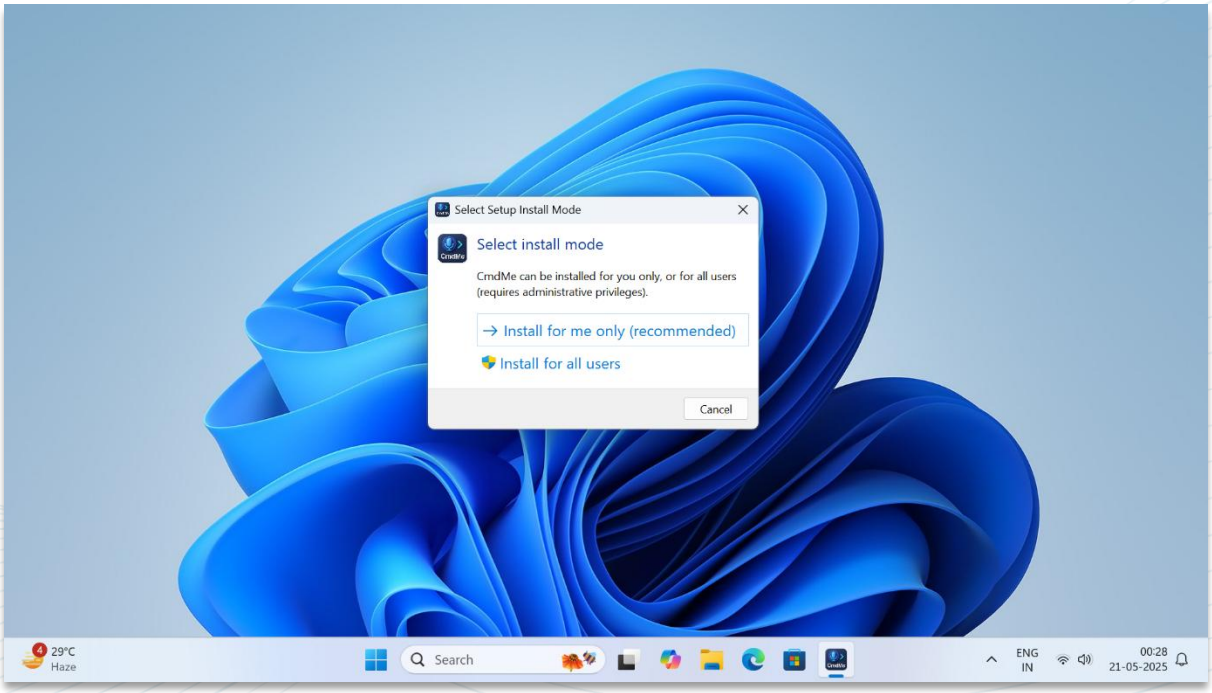
- Locate the CmdMe_Setup.exe file in your system.
- Double-click the file to begin the installation process.



Screenshot 1: *Setup file in File Explorer*

5.2 User Access Selection

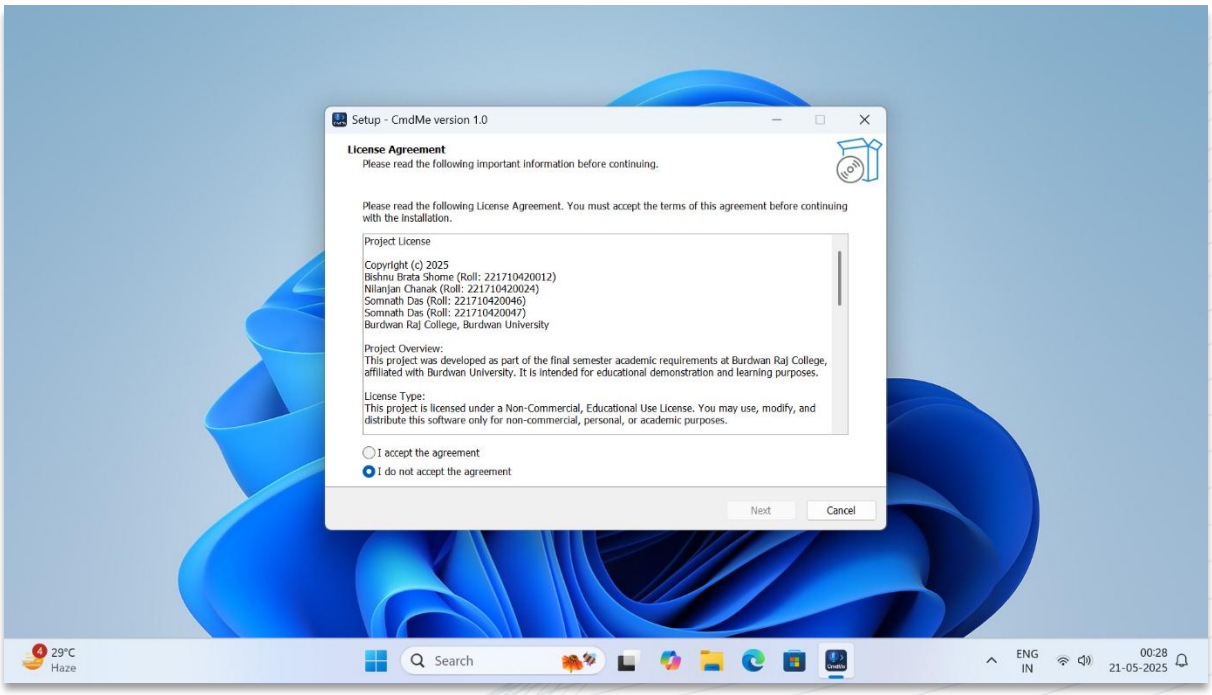
- A pop-up will ask whether to install for "All Users" or "Current User (Recommended)".
- Choose "Current User" unless administrative access is required.



Screenshot 2: *All Users / Current User dialog*

5.3 License Agreement

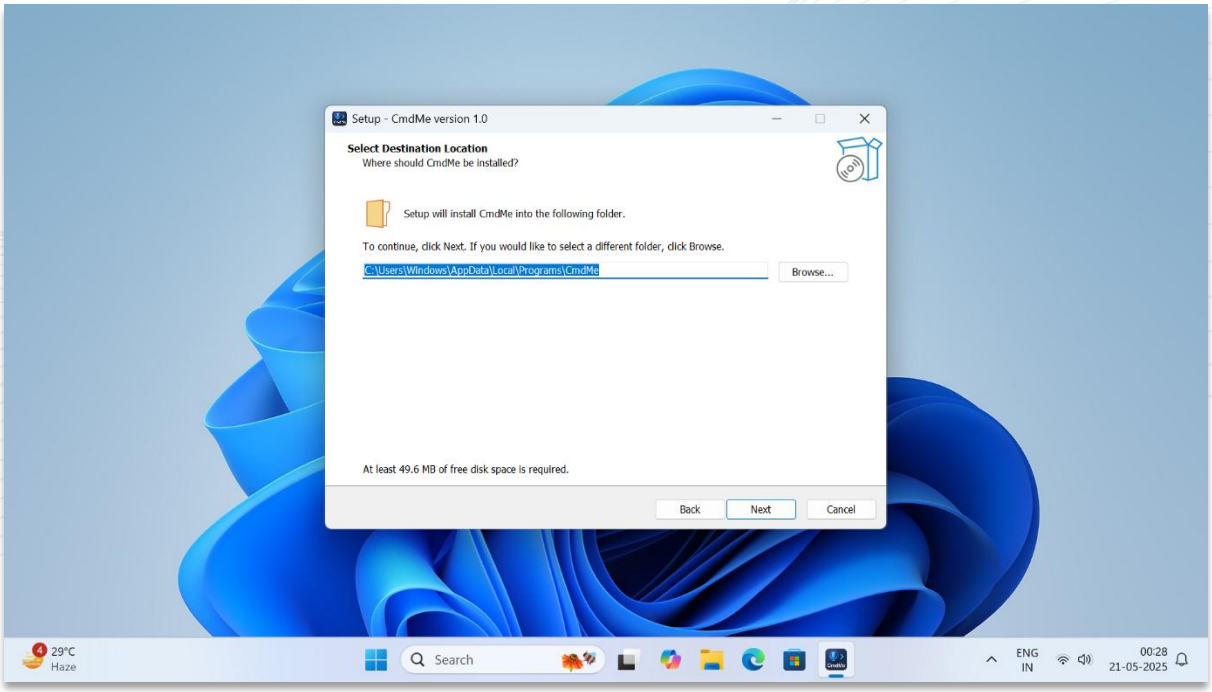
- The setup displays a license agreement.
- Read and accept the agreement to continue.

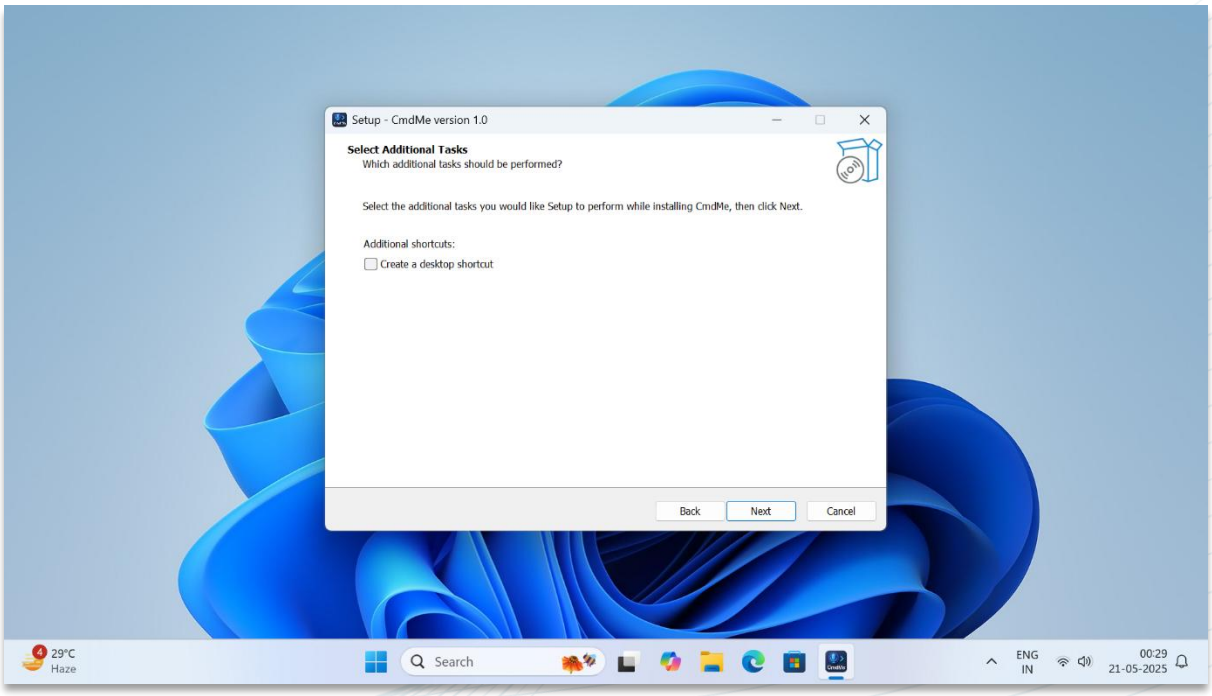


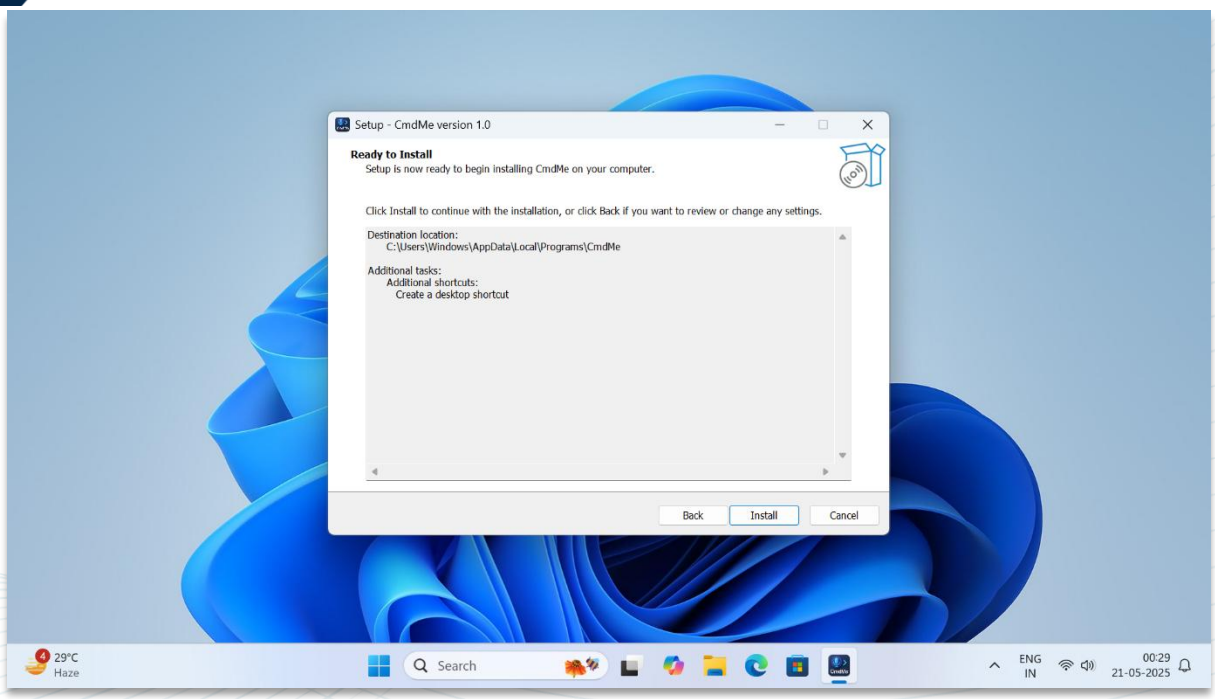
Screenshot 3: *License Agreement page*

5.4 Select Installation Options

- Choose additional tasks like:
    - Create a desktop icon
    - Add to start menu
- Click “Next” to proceed.



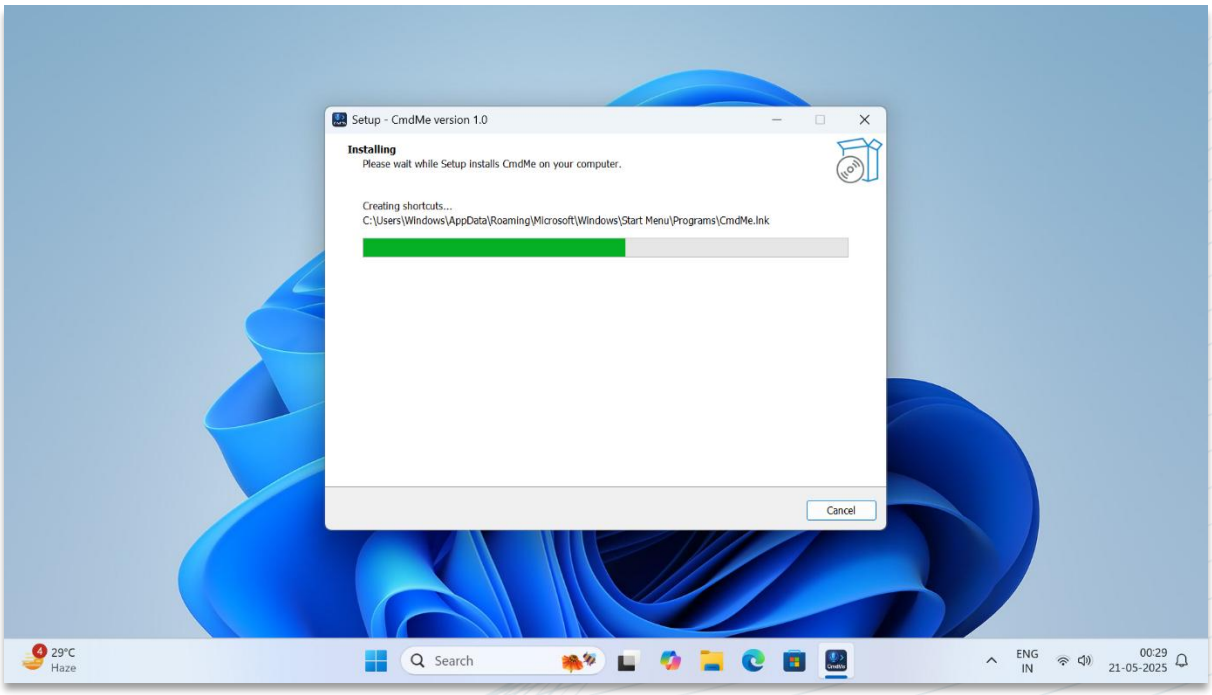




Screenshot 4, 5, 6 : *Additional tasks - desktop icon*

5.5 Installation Progress

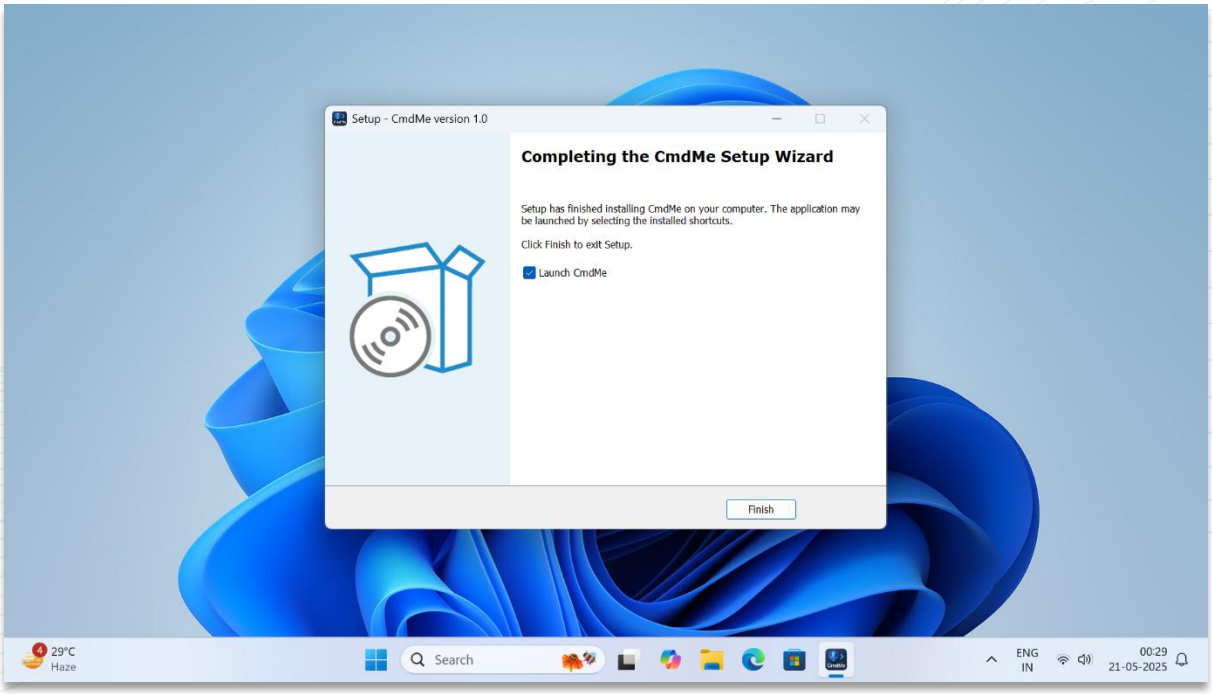
- The setup installs all required files.
- This process may take a few seconds.



Screenshot 7 : *Installation progress*

5.6 Completion

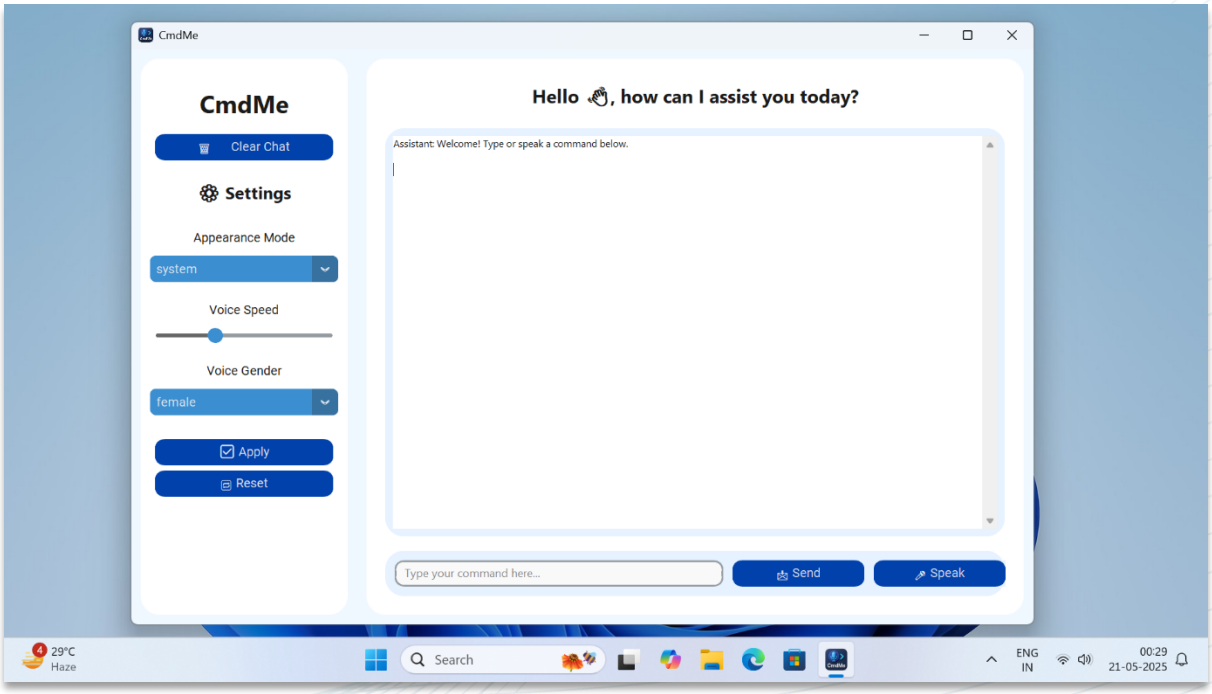
- Once the setup is complete, you can choose to launch **CmdMe** immediately.
- Click “Finish” to close the wizard.



Screenshot 8 : *Setup finished with “Run CmdMe” checked*

5.7 Lunching CmdMe

Once the installation process complete, we can lunch the app, the app starts in windowed mode.



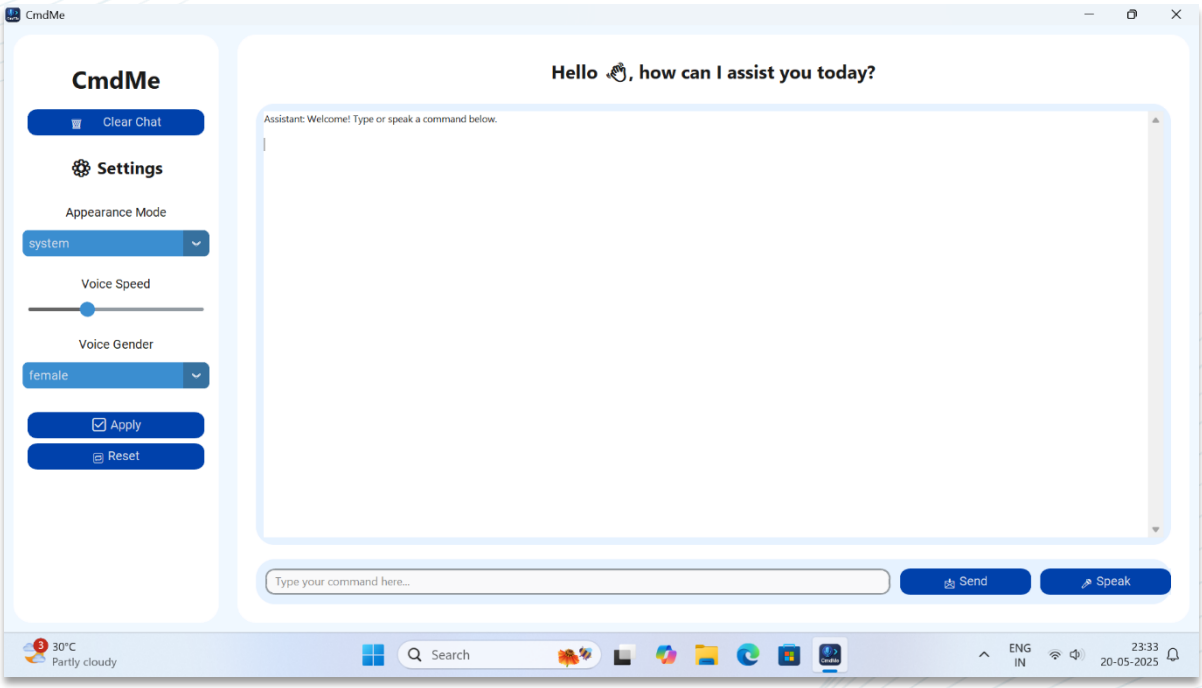
Screenshot 8 : *App started*

6 Screenshots

This section contains screenshots of the **CmdMe – A Voice and Text Command Assistant for Windows** application in action. These visuals demonstrate the graphical user interface, major functionalities, and responses of the system to user input.

6.1 Application Launch

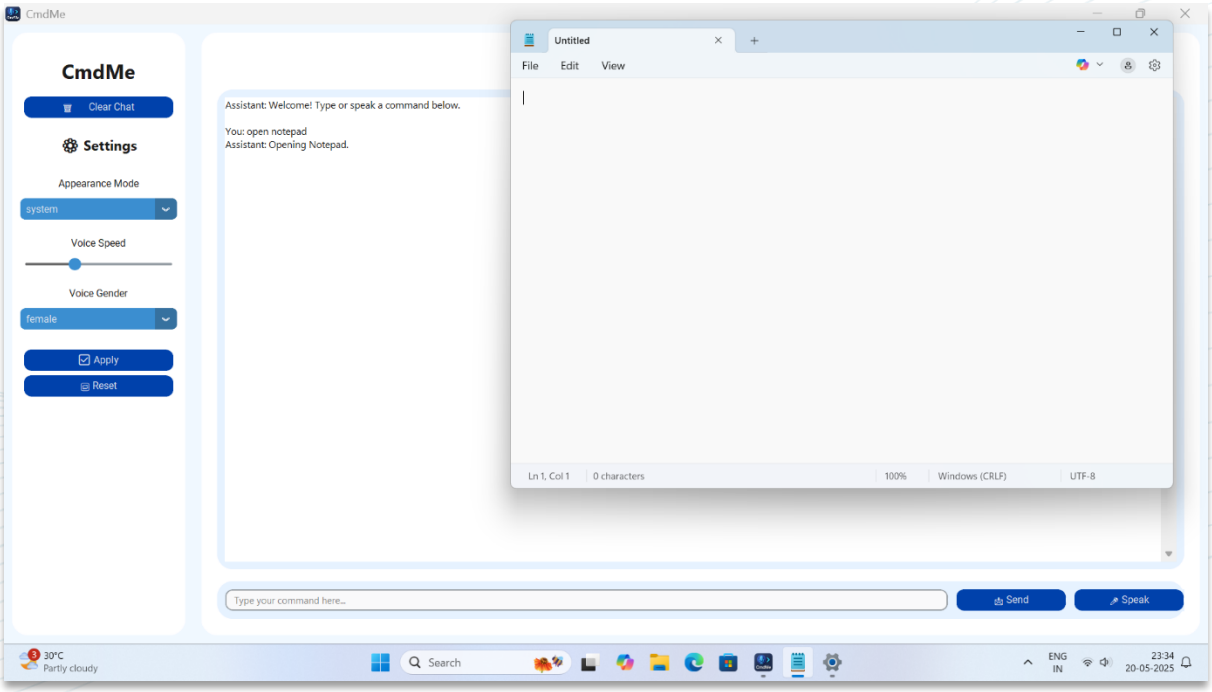
A screenshot showing the application window upon launching, displaying the greeting message and initial layout.



Screenshot 1: *Full application UI after launch*

6.2 Text Command Example

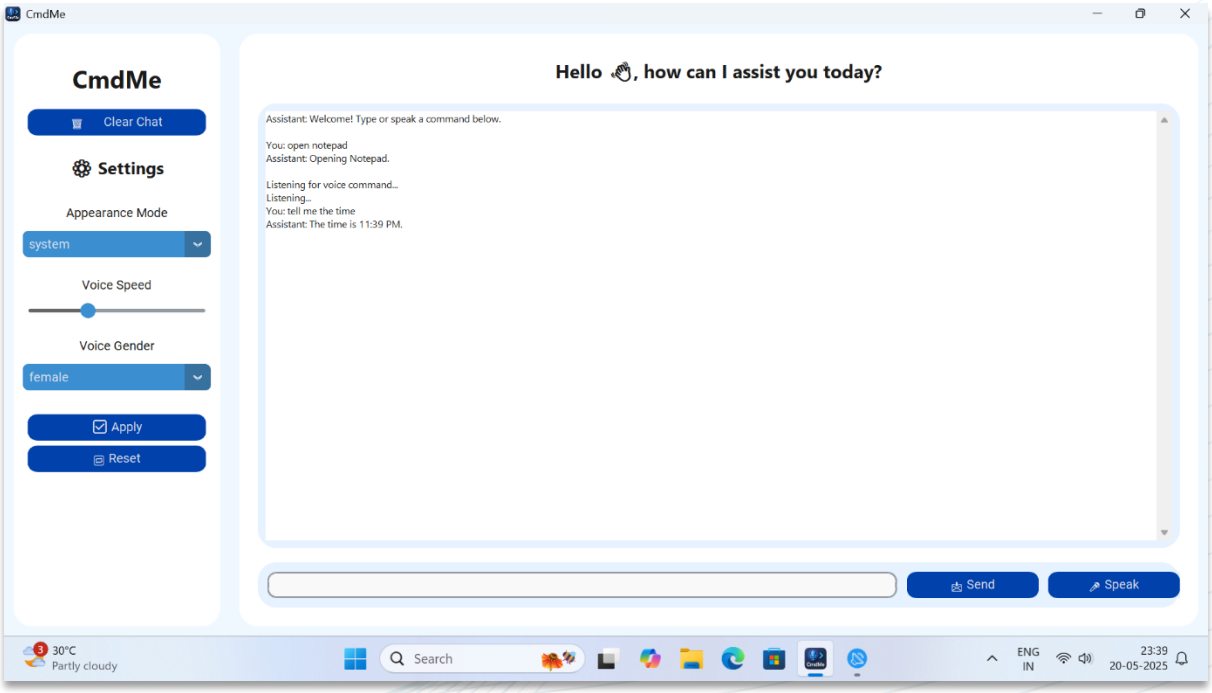
The assistant responds to a typed command such as open notepad.



Screenshot 2 : *Opening Notepad.*

6.3 Voice Command Example

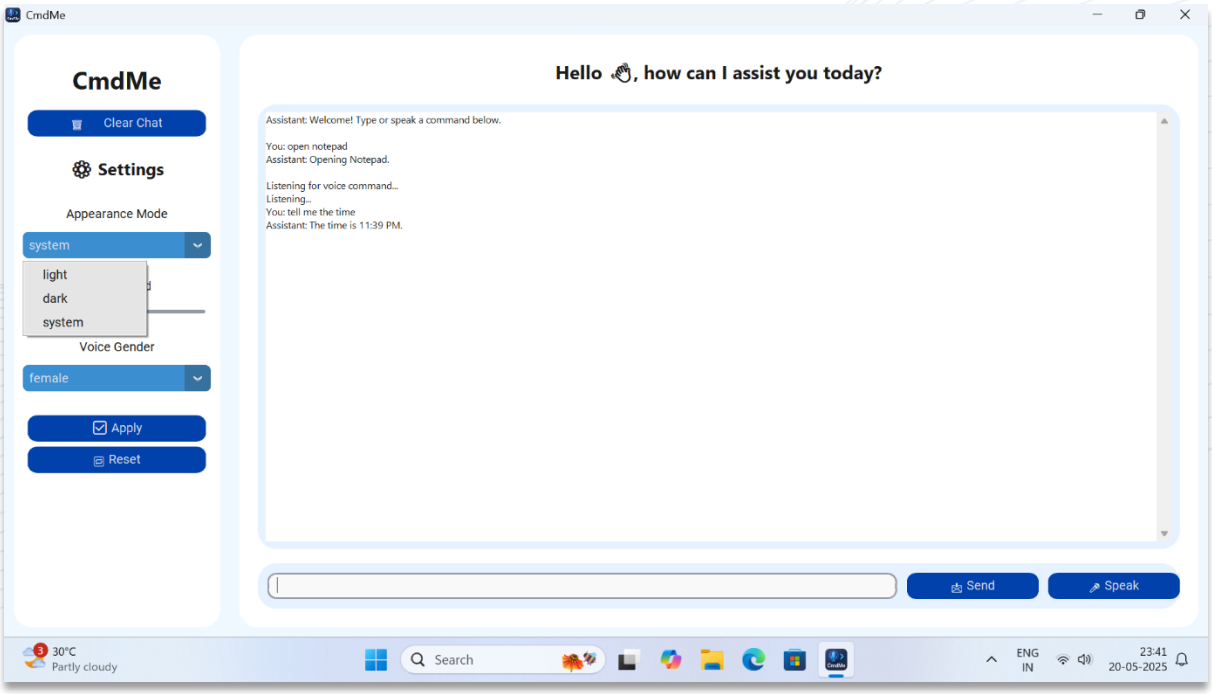
Demonstrates the assistant listening to a voice command and executing it.

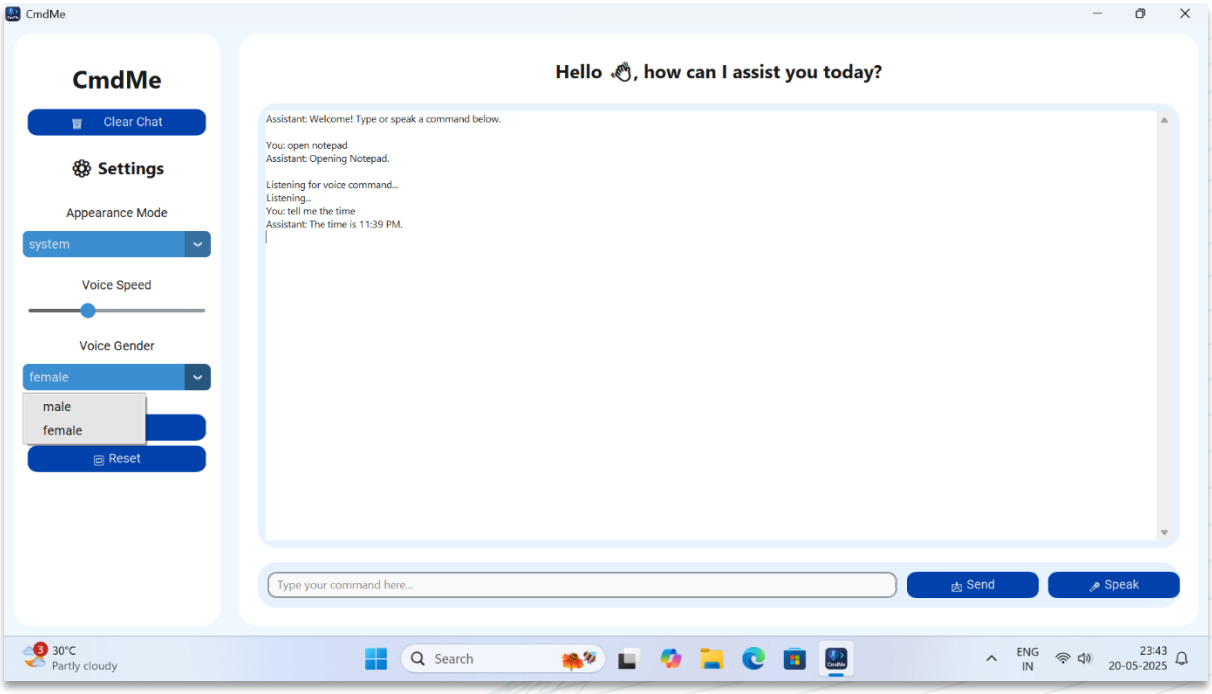


Screenshot 3 : *Listening...*

6.4 Application Settings Panel

This screenshot shows the sidebar with user-configurable settings like appearance mode, voice speed, and gender.

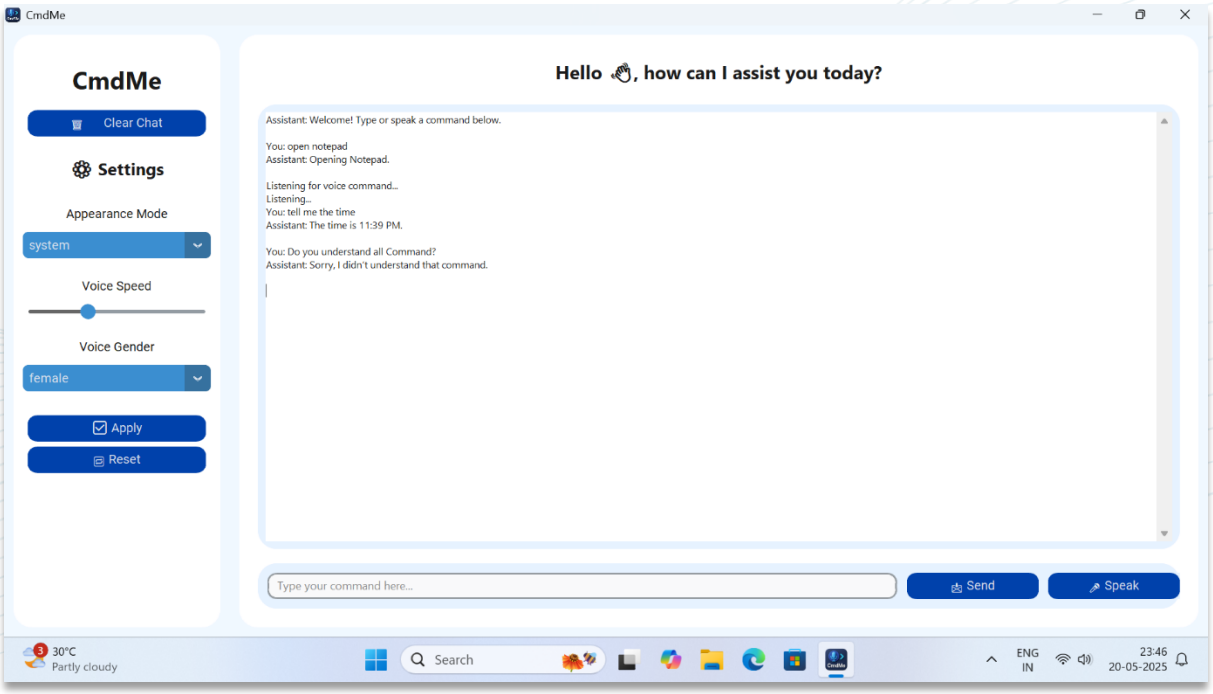




Screenshot 4 and 5 : *Sidebar with sliders and dropdowns visible*

6.5 Error Handling

Shows how the application handles unrecognized commands or errors gracefully.

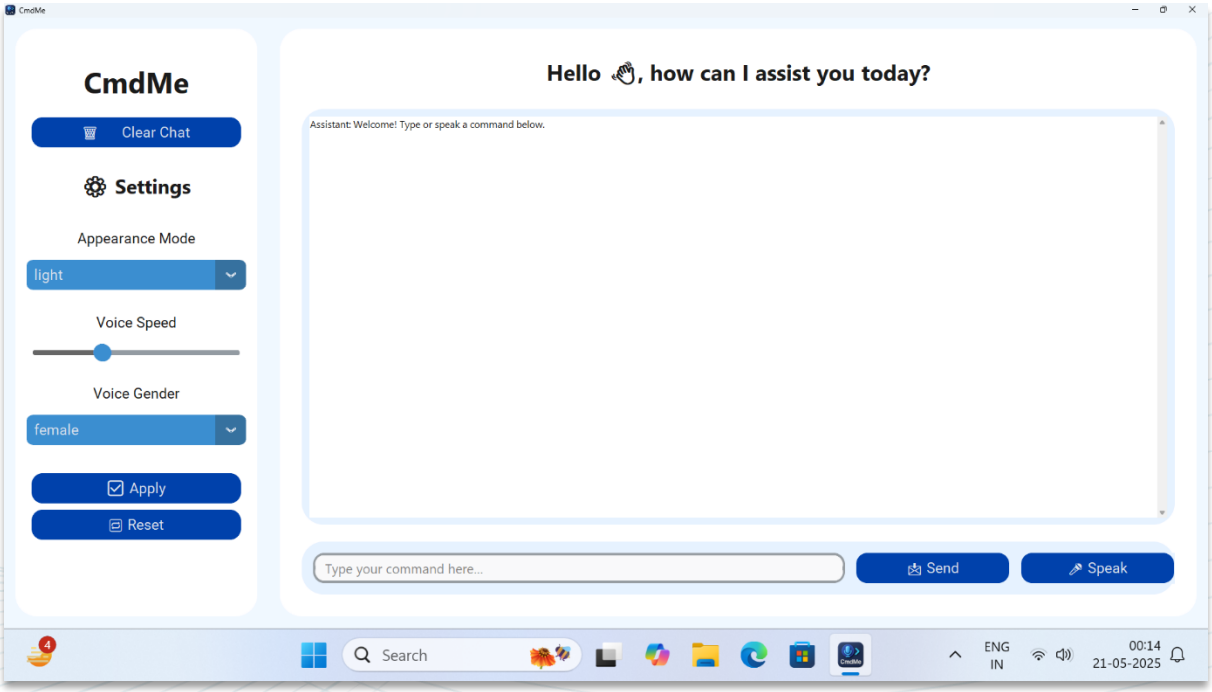


Screenshot 6 : *Message "Sorry, I didn't understand that command."*

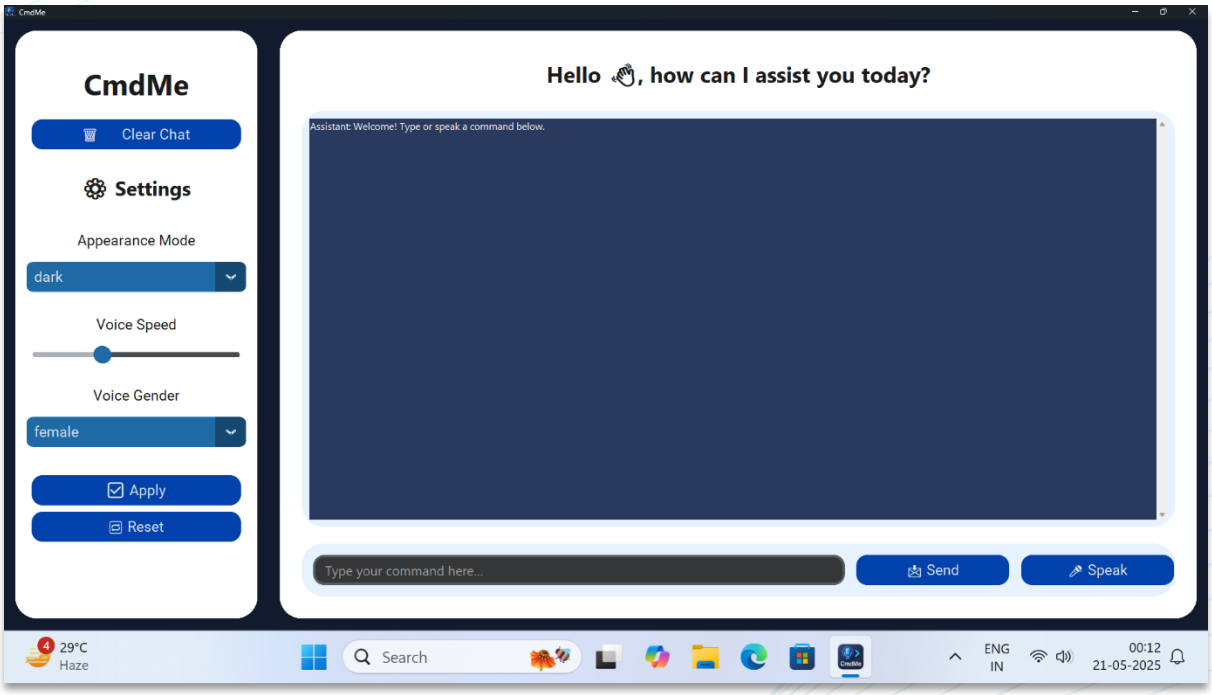
6.6 Theme Support (Dark/Light)

Illustrates the appearance of the app in different themes.

Screenshots



Screenshot 7 : *Light Theme*



Screenshot 7 : *Dark Theme*

7 Testing & Result

The testing phase is essential to ensure that all modules of the system work as expected under various conditions. The objective of this phase was to validate the voice and text command functionality, GUI responsiveness, and overall system performance.

7.1 Test Cases

Below are some test cases used to verify the functionality of different modules of the application:

Test Case	Input	Expected Output	Actual Output	Status
TC01	Open Notepad (Text Command)	Launch Notepad	Notepad launched	Pass
TC02	Open calculator (Voice Command)	Launch Calculator	Calculator launched	Pass
TC03	What is the time?	Return current time	Displays current system time	Pass
TC04	Shutdown	Shutdown system	System begins shutdown	Pass
TC05	Open google.com	Browser opens google.com	Chrome opened google.com	Pass
TC06	Invalid command like Fly to moon	Show fallback message	Displays “I didn’t understand…”	Pass

Voice commands were tested with both clear and noisy environments to ensure robustness.

7.2 Results and Observations

- The system correctly recognized and executed most **voice and text commands** in real-time.

- The **voice recognition** feature was highly accurate in quiet environments but had minor delays or errors in noisy backgrounds.
- The **GUI interface** was responsive and user-friendly, with customizable appearance and voice settings.
- All basic **Windows utilities** (Notepad, Calculator, etc.) opened correctly when available in the system.
- Commands to open **websites and shut down/restart** the system worked as expected.
- **Fallback mechanism** for invalid or unrecognized commands was effective.

7.3 Limitations

- **Internet dependency**: For speech recognition `(recognize_google),` an active internet connection is required.
- **Voice recognition** performance may degrade in **noisy environments** or with **strong accents**.
- The system can only open apps that are **installed and accessible via system path**.
- Limited to **predefined commands** (e.g., specific app names); does not support learning new commands dynamically.
- The assistant currently supports **English only**.

8 Source Code

In this chapter, we present the complete source code for the voice-enabled virtual assistant application, CmdMe. This application is built using Python and the `customtkinter` library for a modern GUI experience. It integrates voice recognition, speech synthesis, and various system automation features. The code brings together all the concepts covered in earlier chapters, such as GUI layout, voice interaction, command processing, and dynamic settings management. The following implementation represents a fully functional desktop assistant that users can interact with through text or voice.

8.1 Full Source Code (Python)

```
# ================================
# Importing Required Libraries
# ================================

import customtkinter as ctk
from tkinter import scrolledtext
import webbrowser
import threading
import pyttsx3
import speech_recognition as sr
import subprocess
import datetime
import os
import sys
import ctypes

# ================================
# Voice Engine Setup
# ================================

engine = pyttsx3.init()
voices = engine.getProperty('voices')

def speak(text):
    """Speaks the given text using TTS engine."""
    engine.say(text)
    engine.runAndWait()
```

```python
def update_voice_settings(rate, gender):
    """Updates voice speed and gender settings."""
    try:
        voice_id = voices[0].id if gender == 'male' else voices[1].id
        engine.setProperty('rate', rate)
        engine.setProperty('voice', voice_id)
    except IndexError:
        pass

# ================================
# Command Handlers
# ================================

def handle_app_opening_commands(command):
    """Handles system app launching commands."""
    apps = {
        "notepad": "notepad.exe",
        "calculator": "calc.exe",
        "paint": "mspaint.exe",
        "command prompt": "cmd.exe",
        "word": "winword.exe",
        "excel": "excel.exe",
        "powerpoint": "powerpnt.exe",
        "chrome": "chrome.exe",
        "vs code": "code",
        "spotify": "spotify.exe"
    }
    for app_name, exe in apps.items():
        if f"open {app_name}" in command:
            try:
                subprocess.Popen(exe)
                return f"Opening {app_name.title()}."
            except FileNotFoundError:
                return f"Sorry, {app_name} is not installed."
    return None

def handle_website_command(command):
    """Handles website opening commands."""
    if "open" in command and ".com" in command:
        words = command.split()
        for word in words:
            if ".com" in word:
                url = word if word.startswith("http") else f"https://{word}"
                webbrowser.open(url)
                return f"Opening {word}."
    return None

def handle_time_date_commands(command):
    """Handles time and date-related queries."""
    now = datetime.datetime.now()
```

```
    if "time" in command:
        return f"The time is {now.strftime('%I:%M %p')}."
    elif "date" in command:
        return f"Today's date is {now.strftime('%B %d, %Y')}."
    return None

def handle_system_commands(command):
    """Handles system-level shutdown and restart."""
    if "shutdown" in command:
        os.system("shutdown /s /t 1")
        return "Shutting down the system."
    elif "restart" in command:
        os.system("shutdown /r /t 1")
        return "Restarting the system."
    return None

# ================================
# Command Processor
# ================================

def process_command(command, chat_display):
    """Processes and delegates the command to the appropriate handler."""
    command = command.lower()
    handlers = [
        handle_app_opening_commands,
        handle_website_command,
        handle_time_date_commands,
        handle_system_commands
    ]
    for handler in handlers:
        response = handler(command)
        if response:
            _respond(chat_display, response)
            return
    _respond(chat_display, "Sorry, I didn't understand that command.")

def _respond(chat_display, message):
    """Displays and speaks the assistant's response."""
    chat_display.insert('end', f"Assistant: {message}\n\n")
    chat_display.see('end')
    speak(message)

# ================================
# Voice Command Listener
# ================================

recognizer = sr.Recognizer()

def listen_voice(chat_display):
    """Captures and processes voice input from the user."""
```

```
    def listen():
        chat_display.insert('end', "Listening...\n")
        chat_display.see('end')
        try:
            with sr.Microphone() as source:
                recognizer.adjust_for_ambient_noise(source)
                audio = recognizer.listen(source)
                command = recognizer.recognize_google(audio)
                chat_display.insert('end', f"You: {command}\n")
                chat_display.see('end')
                process_command(command, chat_display)
        except sr.UnknownValueError:
            _respond(chat_display, "Sorry, I could not understand your
speech.")
        except sr.RequestError:
            _respond(chat_display, "Speech recognition service error.")
        except Exception as e:
            _respond(chat_display, f"An unexpected error occurred: {e}")
    threading.Thread(target=listen, daemon=True).start()

# ================================
# UI Themes (Light & Dark)
# ================================

LIGHT_THEME = {
    "bg_color": "#f0f8ff",
    "frame_color": "#ffffff",
    "widget_color": "#e6f2ff",
    "text_color": "#1a1a1a",
    "chat_bg": "#ffffff",
    "chat_fg": "#1a1a1a",
    "button_fg": "#0041AB",
    "button_hover": "#03358c"
}
DARK_THEME = {
    "bg_color": "#121b2e",
    "frame_color": "#1c2a4a",
    "widget_color": "#2a3a5f",
    "text_color": "#f0f4ff",
    "chat_bg": "#2a3a5f",
    "chat_fg": "#ffffff",
    "button_fg": "#0041AB",
    "button_hover": "#03358c"
}

# ================================
# Main Application GUI Class
# ================================

class AssistantApp(ctk.CTk):
```

```
    def __init__(self):
        super().__init__()
        self.title("CmdMe")
        self.geometry("1000x700")

        # Setup Taskbar Icon
        try:
            ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID(u"
mycompany.cmdme")
        except Exception:
            pass

        # App Icon
        if getattr(sys, 'frozen', False):
            base_path = sys._MEIPASS
        else:
            base_path = os.path.dirname(__file__)
        icon_path = os.path.join(base_path, "icon.ico")
        try:
            self.iconbitmap(icon_path)
        except Exception:
            pass

        # Default Settings
        self.settings = {
            "appearance_mode": "system",
            "voice_rate": 150,
            "voice_gender": "female"
        }

        ctk.set_appearance_mode(self.settings["appearance_mode"])
        self._apply_theme()
        self.grid_columnconfigure(1, weight=1)
        self.grid_rowconfigure(0, weight=1)

        self._build_sidebar()
        self._build_main()
        update_voice_settings(self.settings["voice_rate"],
self.settings["voice_gender"])


    # ====================
    # Theme and Settings
    # ====================

    def _apply_theme(self):
        """Applies light or dark theme."""
        mode = self.settings.get("appearance_mode", "system")
        self.theme = DARK_THEME if (mode == "dark" or
ctk.get_appearance_mode() == "Dark") else LIGHT_THEME
```

```
        self.configure(fg_color=self.theme["bg_color"])

    # ====================
    # Sidebar Layout
    # ====================

    def _build_sidebar(self):
        """Creates the left sidebar for settings."""
        sidebar = ctk.CTkFrame(self, width=180, corner_radius=20,
fg_color=self.theme["frame_color"])
        sidebar.grid(row=0, column=0, sticky="ns", padx=10, pady=10)
        sidebar.grid_propagate(False)

        # Sidebar Header
        ctk.CTkLabel(sidebar, text="CmdMe", font=("Segoe UI", 26, "bold"),
                     text_color=self.theme["text_color"]).pack(pady=(30, 15))

        # Clear Chat Button
        ctk.CTkButton(sidebar, text="🗑 Clear Chat",
fg_color=self.theme["button_fg"],
                      hover_color=self.theme["button_hover"], corner_radius=10,
                      command=self.clear_chat).pack(pady=(0, 20), fill='x',
padx=15)

        # Settings Header
        ctk.CTkLabel(sidebar, text="⚙ Settings", font=("Segoe UI", 18,
"bold"),
                     text_color=self.theme["text_color"]).pack(pady=(0, 10))

        # Settings Controls
        self.setting_widgets = {}
        self._create_dropdown(sidebar, "Appearance Mode",
"appearance_mode", ["light", "dark", "system"])
        self._create_slider(sidebar, "Voice Speed", "voice_rate", 100, 250)
        self._create_dropdown(sidebar, "Voice Gender", "voice_gender",
["male", "female"])

        # Apply/Reset Buttons
        ctk.CTkButton(sidebar, text="☑ Apply",
command=self._apply_app_settings,
                      fg_color=self.theme["button_fg"],
hover_color=self.theme["button_hover"], corner_radius=10).pack(pady=(20,
5), fill='x', padx=15)
        ctk.CTkButton(sidebar, text="🔁 Reset",
command=self._reset_app_settings,
                      fg_color=self.theme["button_fg"],
hover_color=self.theme["button_hover"], corner_radius=10).pack(pady=(0,
10), fill='x', padx=15)
```

```
    # ====================
    # Main Display Area
    # ====================

    def _build_main(self):
        """Builds the main chat and input interface."""
        main_frame = ctk.CTkFrame(self, corner_radius=20,
fg_color=self.theme["frame_color"])
        main_frame.grid(row=0, column=1, sticky="nsew", padx=10, pady=10)

        # Header
        ctk.CTkLabel(main_frame, text="Hello 👋, how can I assist you
today?",
                     font=("Segoe UI", 20, "bold"),
text_color=self.theme["text_color"]).pack(pady=(25, 15))

        # Chat Frame
        chat_frame = ctk.CTkFrame(main_frame, corner_radius=20,
fg_color=self.theme["widget_color"])
        chat_frame.pack(expand=True, fill='both', padx=20, pady=(5, 15))

        self.chat_display = scrolledtext.ScrolledText(
            chat_frame, wrap='word', font=("Segoe UI", 12),
            bg=self.theme["chat_bg"], fg=self.theme["chat_fg"],
            relief='flat', bd=0, insertbackground=self.theme["chat_fg"]
        )
        self.chat_display.pack(expand=True, fill='both', padx=12, pady=12)
        self.chat_display.insert("end", "Assistant: Welcome! Type or speak
a command below.\n\n")

        # Input Section
        input_frame = ctk.CTkFrame(main_frame, height=60, corner_radius=20,
fg_color=self.theme["widget_color"])
        input_frame.pack(fill='x', padx=20, pady=(0, 20))

        self.entry = ctk.CTkEntry(input_frame, placeholder_text="Type your
command here...",
                                  font=("Segoe UI", 12), corner_radius=10)
        self.entry.pack(side='left', fill='x', expand=True, padx=(10, 10),
pady=10)
        self.entry.bind("<Return>", lambda e: self._handle_send())

        # Send & Speak Buttons
        ctk.CTkButton(input_frame, text="📩 Send",
fg_color=self.theme["button_fg"],
                      hover_color=self.theme["button_hover"], corner_radius=10,
                      command=self._handle_send).pack(side='left', padx=(0,
10))
```

```
        ctk.CTkButton(input_frame, text="🎤 Speak",
fg_color=self.theme["button_fg"],
                  hover_color=self.theme["button_hover"], corner_radius=10,
                  command=self._handle_speak).pack(side='left')

    # ====================
    # Settings Controls
    # ====================

    def _create_dropdown(self, parent, label, key, options):
        ctk.CTkLabel(parent, text=label,
text_color=self.theme["text_color"]).pack(pady=(10, 0))
        dropdown = ctk.CTkOptionMenu(parent, values=options, command=lambda
val: self._update_setting(key, val))
        dropdown.set(self.settings[key])
        dropdown.pack(pady=5, fill='x', padx=10)
        self.setting_widgets[key] = dropdown

    def _create_slider(self, parent, label, key, min_val, max_val):
        ctk.CTkLabel(parent, text=label,
text_color=self.theme["text_color"]).pack(pady=(10, 0))
        slider = ctk.CTkSlider(parent, from_=min_val, to=max_val,
number_of_steps=15,
                               command=lambda val: self._update_setting(key,
int(val)))
        slider.set(self.settings[key])
        slider.pack(pady=5, fill='x', padx=10)
        self.setting_widgets[key] = slider

    def _update_setting(self, key, value):
        self.settings[key] = value

    def _apply_app_settings(self):
        ctk.set_appearance_mode(self.settings["appearance_mode"])
        self._apply_theme()
        self._refresh_colors()
        update_voice_settings(self.settings["voice_rate"],
self.settings["voice_gender"])
        speak("Settings applied.")

    def _reset_app_settings(self):
        self.settings = {"appearance_mode": "system", "voice_rate": 150,
"voice_gender": "female"}
        for key, widget in self.setting_widgets.items():
            if isinstance(widget, ctk.CTkOptionMenu):
                widget.set(self.settings[key])
            elif isinstance(widget, ctk.CTkSlider):
                widget.set(self.settings[key])
        self._apply_app_settings()
        speak("Settings reset to default.")
```

```
    def _refresh_colors(self):
        self.chat_display.config(bg=self.theme["chat_bg"],
fg=self.theme["chat_fg"],
                                 insertbackground=self.theme["chat_fg"])
        self.configure(fg_color=self.theme["bg_color"])


    # ====================
    # Input Handlers
    # ====================

    def clear_chat(self):
        self.chat_display.delete("1.0", "end")

    def _handle_send(self):
        text = self.entry.get().strip()
        if text:
            self.chat_display.insert("end", f"You: {text}\n")
            self.chat_display.see("end")
            self.entry.delete(0, 'end')
            process_command(text, self.chat_display)

    def _handle_speak(self):
        self.chat_display.insert("end", "Listening for voice command...\n")
        self.chat_display.see("end")
        listen_voice(self.chat_display)

# ================================
# Application Entry Point
# ================================

if __name__ == "__main__":
    app = AssistantApp()
    app.mainloop()
```

Conclusion:

In the following chapter, each part of the source code will be explained line by line, detailing the logic, modules, and interactions involved in the application. This will help deepen your understanding of the architecture and how all the components come together.

9 Code Explanation

This chapter provides a detailed explanation of the entire source code presented in Chapter 8. Each section of the code is broken down with commentary to help the reader understand how individual components work and how they collectively contribute to the functionality of the CmdMe assistant application.

9.1 Importing Required Libraries

The first step in building any Python-based project is importing the necessary libraries. These libraries provide the tools, functions, and APIs required to implement various features such as GUI creation, voice recognition, system operations, threading, and more.

```
# ================================
# Importing Required Libraries
# ================================
```

This is a comment section used to clearly separate and identify the purpose of the code block. It improves readability and helps in understanding the structure of the code.

```
import customtkinter as ctk
```

- `customtkinter` is an extended version of `tkinter` that provides a modern, customizable user interface.
- It includes advanced widgets and theming capabilities.
- It is imported here with the alias `ctk` for easier usage throughout the code.

```
from tkinter import scrolledtext
```

- This imports the `scrolledtext` widget from the standard `tkinter` library.
- `scrolledtext` is a text box that supports scrolling, useful for displaying conversation history in the assistant's chat interface.

```
import webbrowser
```

- The `webbrowser` module allows the program to open web pages in the user's default browser.
- It's used for handling commands like "open google.com".

```
import threading
```

- threading is used to run multiple operations in parallel without freezing the GUI.
- It allows background tasks like voice recognition to run smoothly while keeping the interface responsive.

```
import pyttsx3
```

- This is a Python library for text-to-speech conversion.
- It enables the assistant to "speak" responses aloud, enhancing interactivity.

```
import speech_recognition as sr
```

- This library is used to convert spoken words into text.
- It supports various speech recognition engines, including Google's speech recognition API.

```
import subprocess
```

- subprocess enables the execution of system-level commands and opening applications.
- It's used in the project to launch programs like Notepad or Chrome via user commands.

```
import datetime
```

- This module allows the application to access and format the current date and time.
- It supports features like "what time is it" or "tell me the date".

```
import os
```

- `os` provides functions for interacting with the operating system.
- It's used for operations such as system shutdown or restart.

```
import sys
```

- The sys module gives access to system-specific parameters and functions.
- In this project, it helps in managing application paths, especially when the application is compiled into an executable.

```
import ctypes
```

- `ctypes` is used here to set the Application User Model ID on Windows.
- This allows the app to show a custom icon in the Windows taskbar, enhancing branding and appearance.

➢ **Summary:**

This set of imports forms the foundation of the CmdMe assistant. Each module has a clear and specific role:

- **UI creation**: `customtkinter, scrolledtext`
- **Voice capabilities**: `pyttsx3, speech_recognition`
- **System control**: `os, subprocess, datetime`
- **Web access**: `webbrowser`
- **Utility and optimization**: `threading, ctypes, sys`

By understanding these libraries, the reader can appreciate how CmdMe brings together various Python capabilities to offer a powerful, interactive assistant.

9.2 Voice Engine Setup

Voice interaction is one of the core features of CmdMe. To enable the assistant to "speak" to users, we use a text-to-speech (TTS) engine powered by the pyttsx3 library. This section of the code sets up the voice engine and provides functions to produce spoken output and customize voice properties like speed and gender.

```
# ================================
# Voice Engine Setup
# ================================

engine = pyttsx3.init()
```

- This initializes the text-to-speech engine provided by the `pyttsx3` module.
- The `init()` function creates an engine object that acts as the interface for all TTS operations.

```
voices = engine.getProperty('voices')
```

- Retrieves a list of all available voices installed on the system.
- These voice profiles may include male, female, or localized accents, depending on the OS configuration.

➢ **Function:** `speak(text):`

```
def speak(text):
    """Speaks the given text using TTS engine."""
    engine.say(text)
    engine.runAndWait()
```

- **Purpose**: To convert a given string of text into audible speech.
- `engine.say(text)`: Queues the text for speech synthesis.
- `engine.runAndWait()`: Processes the speech queue and actually speaks it aloud.
- This function is used throughout the app to vocalize assistant responses.

➢ **Function:** `update_voice_settings(rate, gender)`

```
def update_voice_settings(rate, gender):
    """Updates voice speed and gender settings."""
    try:
        voice_id = voices[0].id if gender == 'male' else voices[1].id
        engine.setProperty('rate', rate)
        engine.setProperty('voice', voice_id)
    except IndexError:
        pass
```

- **Purpose**: Allows the user to adjust the **speed** (rate) and **gender** of the assistant's voice.
- `rate`: The number of words per minute the engine speaks.
- `gender`: A string, either `'male'` or `'female'`, used to choose the appropriate voice.
- The `try` block ensures the code does not crash if a voice is not found (e.g., if only one voice is available).

Voice Selection Logic:

- `voices[0]` is assumed to be the male voice.
- `voices[1]` is assumed to be the female voice.
- This assumption works on many systems but could vary depending on installed voice packs.

➢ **Summary:**

This section enables CmdMe to:

- Speak responses in real-time.
- Let users personalize the voice experience with adjustable speed and gender.

By encapsulating TTS functionality in `speak()` and `update_voice_settings()`, the application separates core logic from configuration, allowing easy modifications and future enhancements (like adding language selection or multiple voice profiles).

9.3 Command Handlers

CmdMe acts upon user instructions by analyzing the content of the given command and executing corresponding actions—like launching an app, opening a website, telling the time, or shutting down the system. This section of code contains individual **handler functions**, each designed to process a specific category of command.

➢ `handle_app_opening_commands(command)`

```
# ================================
# Command Handlers
# ================================
```

```
def handle_app_opening_commands(command):
    """Handles system app launching commands."""
```

- This function scans for keywords to open various pre-defined system applications like Notepad or Chrome.
- It returns a user-friendly confirmation string if an app is opened, or a failure message if the app isn't installed.

```
    apps = {
        "notepad": "notepad.exe",
        "calculator": "calc.exe",
        "paint": "mspaint.exe",
        "command prompt": "cmd.exe",
        "word": "winword.exe",
        "excel": "excel.exe",
        "powerpoint": "powerpnt.exe",
        "chrome": "chrome.exe",
        "vs code": "code",
        "spotify": "spotify.exe"
    }
```

- A dictionary that maps friendly app names (used in user speech) to their actual executable commands.

```
    for app_name, exe in apps.items():
        if f"open {app_name}" in command:
```

- Iterates through the dictionary to find if the user's command contains a phrase like `"open notepad"`.

```
            try:
                subprocess.Popen(exe)
                return f"Opening {app_name.title()}."
```

- If matched, attempts to launch the app using `subprocess.Popen()`, which starts a new process in the background.

```
            except FileNotFoundError:
                return f"Sorry, {app_name} is not installed."
```

- Catches errors if the executable is not found, preventing a crash and instead providing a graceful error message.

```
    return None
```

- If no matching app is found, returns `None`, signaling no action was taken.

➢ `handle_website_command(command)`

```
def handle_website_command(command):
    """Handles website opening commands."""
```

- Parses a command to detect and open a web address (e.g., `"open google.com"`).

```
    if "open" in command and ".com" in command:
        words = command.split()
```

- Ensures the command has both “open” and a “.com” domain keyword. Splits the sentence into individual words.

```
        for word in words:
            if ".com" in word:
                url = word if word.startswith("http") else f"https://{word}"
                webbrowser.open(url)
                return f"Opening {word}."
```

- Finds the website string, adds `https://` if missing, and opens it using the default browser.
- Returns a success message once done.

```
    return None
```

- Again, returns `None` if no valid URL is found in the command.

➢ `handle_time_date_commands(command)`

```
def handle_time_date_commands(command):
    """Handles time and date-related queries."""
    now = datetime.datetime.now()
```

- Gets the current system time and date.

```
    if "time" in command:
        return f"The time is {now.strftime('%I:%M %p')}."
```

- If the user asks for the time, returns it in 12-hour format with AM/PM.

```
    elif "date" in command:
        return f"Today's date is {now.strftime('%B %d, %Y')}."
```

- If the user asks for the date, returns a full human-readable date.

```
    return None
```

- Returns None if neither time nor date is requested.

➢ `handle_system_commands(command)`

```
def handle_system_commands(command):
    """Handles system-level shutdown and restart."""
```

- These are more sensitive commands for controlling the operating system directly.

```
    if "shutdown" in command:
        os.system("shutdown /s /t 1")
        return "Shutting down the system."
```

- Triggers a system shutdown with 1-second delay.

```
    elif "restart" in command:
        os.system("shutdown /r /t 1")
        return "Restarting the system."
```

- Restarts the system with the same 1-second delay.

```
    return None
```

- Returns `None` if neither command is detected.

➢ **Summary:**

Each handler in this section is **specialized** and **independent**, following the **single-responsibility principle**. This modular approach helps keep the code clean and extensible. The command handlers enable CmdMe to:

- Launch commonly used applications.
- Open websites via browser.
- Answer time/date-related queries.
- Perform system-level tasks like shutdown and restart.

In the next section, we'll explore how these handlers are all brought together under a single **command processing function**.

9.4 Command Processor

At the core of CmdMe's intelligence is the **command processor**. This function is responsible for taking the raw voice-to-text command from the user, determining what it means, and routing it to the appropriate handler function we discussed earlier.

➢ `process_command(command, chat_display)`

```
# ================================
# Command Processor
# ================================

def process_command(command, chat_display):
    """Processes and delegates the command to the appropriate handler."""
```

- This function receives two inputs:

- **command**: The spoken input converted to text.
- **chat_display**: A GUI component (likely a scrolled text area) where responses will be shown.

```
command = command.lower()
```

- Normalizes the command to lowercase to ensure consistent keyword matching (since user input could be in any case format).

```
handlers = [
    handle_app_opening_commands,
    handle_website_command,
    handle_time_date_commands,
    handle_system_commands
]
```

- A list of handler functions defined earlier, each responsible for a category of commands.
- This makes the function modular and easily extendable—you could add more handlers here in the future.

```
for handler in handlers:
    response = handler(command)
```

- Iterates through the handler list, sending the command to each one.
- Each handler returns a response **only** if it was able to process the command.

```
    if response:
        _respond(chat_display, response)
        return
```

- The moment a valid response is received, the assistant:
  - Displays the response in the GUI via `_respond()`.
  - Speaks the response using text-to-speech.
- Then it **immediately returns**, ensuring only one handler acts on a command (no overlapping actions).

```
_respond(chat_display, "Sorry, I didn't understand that command.")
```

- If **none** of the handlers return a response, a fallback message is triggered.

➢ `_respond(chat_display, message)`

```
def _respond(chat_display, message):
    """Displays and speaks the assistant's response."""
```

- This helper function manages both GUI display and voice output for assistant replies.

```
chat_display.insert('end', f"Assistant: {message}\n\n")
```

- Appends the assistant's reply to the chat interface for visual feedback.

```
chat_display.see('end')
```

- Automatically scrolls the chat window to the most recent message.

```
speak(message)
```

- Uses the previously defined `speak()` function to vocalize the assistant's response, completing the interaction loop.

➢ **Summary:**

The `process_command` function serves as the **central hub** of the voice assistant's logic. It's responsible for:

- Delegating to the correct command handler.
- Ensuring the assistant reacts appropriately.
- Providing both visual and auditory feedback.

By keeping this function clean and modular, CmdMe remains easy to maintain and scalable—perfect for future enhancements like AI integration or NLP-based parsing.

9.5 Voice Recognition and Listening

In this section, we will explore how CmdMe listens to your voice, interprets your commands, and initiates a response. This is a critical part of the assistant, as it's the entry point for all spoken user interactions.

➢ **Speech Recognition Setup:**

```
# ================================
# Voice Command Listener
# ================================

recognizer = sr.Recognizer()
```

- This line creates an instance of the `Recognizer` class from the `speech_recognition` module.
- This object is responsible for:
  - Listening to audio input.
  - Converting it to text using speech-to-text APIs.

➢ `listen_voice(chat_display)`

```
def listen_voice(chat_display):
    """Captures and processes voice input from the user."""
```

- This function defines the listening behavior and starts the voice capture process.
- `chat_display` is used to display status messages and recognized speech in the GUI.

➢ **Nested `listen()` Function:**

Inside `listen_voice`, a **nested function** is defined and executed in a background thread:

```
    def listen():
```

- This approach allows the GUI to remain responsive while listening occurs in the background.

```
        chat_display.insert('end', "Listening...\n")
        chat_display.see('end')
```

- Updates the GUI to inform the user that the assistant is actively listening.

➢ **Capturing Audio:**

```
        try:
            with sr.Microphone() as source:
                recognizer.adjust_for_ambient_noise(source)
```

- Opens the microphone as the audio source.
- The `adjust_for_ambient_noise()` method calibrates for background noise, improving accuracy.

```
                audio = recognizer.listen(source)
```

- Listens to the user's speech and records it into an audio object.

➢ **Recognizing Speech:**

```
                command = recognizer.recognize_google(audio)
```

- Converts the recorded audio into text using Google's speech recognition API.

```
                chat_display.insert('end', f"You: {command}\n")
                chat_display.see('end')
```

- Displays the recognized command in the GUI, simulating a chat-like experience.

```
                process_command(command, chat_display)
```

- Passes the recognized command to the processor we explored earlier for interpretation and execution.

➢ **Error Handling:**

The assistant is designed to handle various common issues gracefully:

```
    except sr.UnknownValueError:
        _respond(chat_display, "Sorry, I could not understand your
speech.")
```

- Triggered when the speech was unclear or not understood.

```
    except sr.RequestError:
        _respond(chat_display, "Speech recognition service error.")
```

- Happens when there's a connectivity issue or problem with the speech recognition API.

```
    except Exception as e:
        _respond(chat_display, f"An unexpected error occurred: {e}")
```

- A catch-all for unexpected errors that might occur during the listening or recognition process.

➢ **Running in a Background Thread:**

```
threading.Thread(target=listen, daemon=True).start()
```

- Starts the `listen()` function in a separate **daemon thread**.
- This ensures the GUI won't freeze while the assistant listens.
- The thread is marked as `daemon`, meaning it will exit automatically when the application closes.

➢ **Summary:**

The `listen_voice()` function transforms CmdMe from a passive application into an interactive voice assistant. It:

- Actively listens for voice commands.
- Handles recognition and errors with user-friendly feedback.
- Maintains GUI responsiveness through multithreading.

This modular setup allows CmdMe to feel fast, responsive, and conversational—core aspects of any intelligent voice assistant.

9.6 User Interface Themes – Light & Dark Modes

A pleasant and functional user interface is essential for any modern application. CmdMe supports both light and dark themes to enhance visual comfort and accessibility. In this

section, we'll explore how themes are defined and how these color palettes affect the application's look and feel.

🎨 What is a Theme?

A **theme** is essentially a collection of colors and visual settings applied across the user interface to create a consistent appearance. CmdMe defines two themes:

- **LIGHT_THEME**: For well-lit environments.
- **DARK_THEME**: For low-light or night-time use.

These themes are implemented using Python dictionaries, allowing for easy updates and access across the application.

🔆 Light Theme Definition:

```
# ================================
# UI Themes (Light & Dark)
# ================================

LIGHT_THEME = {
   "bg_color": "#f0f8ff",
   "frame_color": "#ffffff",
   "widget_color": "#e6f2ff",
   "text_color": "#1a1a1a",
   "chat_bg": "#ffffff",
   "chat_fg": "#1a1a1a",
   "button_fg": "#0041AB",
   "button_hover": "#03358c"
}
```

- `bg_color`: The background color of the main application window. A soft **AliceBlue** shade improves readability.
- `frame_color`: The color of internal containers or panels, set to white for a clean separation.
- `widget_color`: A slightly shaded blue background for buttons, entries, and other widgets.
- `text_color`: Standard dark gray text to ensure contrast and legibility.
- `chat_bg`: The background color of the chat display area—kept white for a paper-like feel.
- `chat_fg`: Chat text color, matching `text_color`.
- `button_fg`: Foreground color (typically text) of buttons, using a rich blue tone for clarity.
- `button_hover`: A darker blue shade shown when a user hovers over a button, adding interactivity.

🌑 Dark Theme Definition:

```
DARK_THEME = {
```

```
    "bg_color": "#121b2e",
    "frame_color": "#1c2a4a",
    "widget_color": "#2a3a5f",
    "text_color": "#f0f4ff",
    "chat_bg": "#2a3a5f",
    "chat_fg": "#ffffff",
    "button_fg": "#0041AB",
    "button_hover": "#03358c"
}
```

- `bg_color`: A deep navy blue that provides a dark but non-black backdrop.
- `frame_color`: Slightly lighter than the background for visual structure.
- `widget_color`: Muted steel blue for widget backgrounds, ensuring they stand out.
- `text_color`: A soft off-white for reading in the dark without harsh contrast.
- `chat_bg` and `chat_fg`: These maintain the same theme logic for the chat interface.
- `button_fg` and `button_hover`: Same blues as the light theme, maintaining consistency.

🛠️ **Why Use Dictionaries?**

By storing theme values in dictionaries:

- Themes can be applied dynamically.
- It simplifies updating or adding new themes in the future.
- It makes your GUI code cleaner, as you can use `theme["bg_color"]` rather than hardcoding colors everywhere.

➢ **Summary:**

With these two theme dictionaries, CmdMe is visually adaptable to different lighting environments and user preferences. They act as the **style guide** of your application, keeping the look unified and user-friendly.

9.7 Main Application Class – AssistantApp

The core of our application's graphical user interface is encapsulated in the `AssistantApp` class. This class handles the main window, appearance settings, voice engine configuration, and layout initialization. It is a subclass of `CTk`, which is a themed version of `Tkinter` provided by the `customtkinter` library.
In this section, we will walk through this class line-by-line to understand how the main interface of CmdMe is structured.

🧱 **Class Definition**

```
# ================================
# Main Application GUI Class
# ================================

class AssistantApp(ctk.CTk):
```

We define a class **AssistantApp** that inherits from **ctk.CTk**. This gives us access to all functionalities of a custom-themed **Tkinter** window.

🏗️ Constructor – __init__

```
   def __init__(self):
      super().__init__()
```

- The **__init__** method initializes the GUI class.
- **super().__init__()** ensures the parent class (**CTk**) is properly initialized, setting up the main application window.

🖼️ Window Title and Size

```
      self.title("CmdMe")
      self.geometry("1000x700")
```

- Sets the title of the window to "CmdMe".
- **geometry** sets the window size to **1000 pixels wide** and **700 pixels tall**.

🧭 Windows Taskbar Integration

```
      # Setup Taskbar Icon
      try:
         ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID(u"
mycompany.cmdme")
      except Exception:
         pass
```

- This snippet improves how the app appears in the Windows taskbar.
- It assigns a custom app ID (**mycompany.cmdme**), making the app icon and grouping behave like a native app.
- We use a **try–except** block to handle errors on non-Windows systems gracefully.\

🖌️ Application Icon Setup

```
      # App Icon
      if getattr(sys, 'frozen', False):
         base_path = sys._MEIPASS
      else:
         base_path = os.path.dirname(__file__)
      icon_path = os.path.join(base_path, "icon.ico")
```

- Checks if the app is running as a bundled executable (like from PyInstaller).
- **sys._MEIPASS** is used by **PyInstaller** to reference bundled files.

- Builds the full path to the icon file: icon.ico.

```
try:
    self.iconbitmap(icon_path)
except Exception:
    pass
```

- Attempts to apply the custom icon to the window.
- The `try-except` block ensures the app doesn't crash if the icon is missing.

⚙️ **Application Settings Initialization**

```
# Default Settings
self.settings = {
    "appearance_mode": "system",
    "voice_rate": 150,
    "voice_gender": "female"
}
```

- A dictionary holding default settings for:
    - Appearance (`light`, `dark`, or `system`)
    - Voice speed (words per minute)
    - Voice gender (`male` or `female`)

🎨 **Applying Appearance Mode and Theme**

```
ctk.set_appearance_mode(self.settings["appearance_mode"])
self._apply_theme()
```

- Sets the theme based on system or user choice.
- Calls `_apply_theme()` to apply specific color configurations (defined earlier as `LIGHT_THEME` or `DARK_THEME`).

🧩 **Grid Configuration**

```
self.grid_columnconfigure(1, weight=1)
self.grid_rowconfigure(0, weight=1)
```

- Prepares the layout manager to ensure the main content area expands properly.
- Assigns weight to rows and columns so widgets resize with the window.

🧱 **Build Sidebar and Main Chat Area**

```
self._build_sidebar()
self._build_main()
```

- These functions construct the sidebar and main area where the user will interact with the assistant. We'll explore these in detail in the next section.

Set Initial Voice Settings

```
    update_voice_settings(self.settings["voice_rate"],
self.settings["voice_gender"])
```

- Initializes the text-to-speech engine using the default voice settings.
- This ensures that CmdMe can respond vocally as soon as it launches.

➢ **Summary:**

The `AssistantApp` class serves as the control center of the application. It sets up the window, icon, voice settings, themes, and layout. By using `customtkinter`, we ensure the app is visually modern and responsive.

9.8 Theme and Settings Management

One of the benefits of using `customtkinter` is its built-in support for light and dark themes. In this section, we examine the `_apply_theme` method, which enables dynamic switching between these themes based on user or system preferences.

This functionality is crucial for delivering a modern user experience, as users today expect apps to adapt to their light or dark environment preferences.

`_apply_theme` Method

```
  def _apply_theme(self):
```

This method is a **private utility** method (by convention, not enforcement), indicated by the underscore `_`. It applies the appropriate theme — either **light** or **dark** — based on user settings or system configuration.

Determine Appearance Mode

```
    """Applies light or dark theme."""
    mode = self.settings.get("appearance_mode", "system")
```

- Retrieves the preferred appearance mode from the `settings` dictionary.
- If no mode is set explicitly, it defaults to `"system"`, meaning it will match the operating system's theme.

Select Appropriate Theme Colors

```
    self.theme = DARK_THEME if (mode == "dark" or
ctk.get_appearance_mode() == "Dark") else LIGHT_THEME
```

- Uses a conditional expression to determine which theme dictionary to apply:

- If the mode is explicitly set to `"dark"` **OR**
- If the customtkinter appearance mode is detected as `"Dark"`, then it applies the `DARK_THEME`.

- Otherwise, it defaults to `LIGHT_THEME`.

The theme dictionaries (`DARK_THEME` and `LIGHT_THEME`) contain predefined color values for different UI components such as background, buttons, and text.

🖼️ **Apply Background Color**

```
        self.configure(fg_color=self.theme["bg_color"])
```

- `self.configure()` sets the background color (`fg_color`) of the main window using the selected theme.
- This is the first step in applying the complete theme across the application. Other UI elements (buttons, frames, labels, etc.) will also use the same theme dictionary to style themselves consistently.

➢ **Summary:**

The `_apply_theme` method is responsible for determining and applying the visual theme of the application. It pulls the desired mode from user settings (or the system) and uses predefined color schemes to update the app's background.

This modular approach — separating theming logic from GUI building — makes the code cleaner, more maintainable, and extensible. For instance, adding a new theme (like "blue mode" or "high contrast") would be as simple as defining a new theme dictionary and referencing it here.

9.9 Sidebar Layout: Settings and Controls

The sidebar is a vital part of the user interface in the CmdMe assistant. It provides essential settings and controls that allow users to personalize their experience, such as switching themes, adjusting voice settings, or resetting configurations. This section walks through how the sidebar is created and how various widgets are integrated into it.

🔧 `_build_sidebar()` **Method**

```
    # ====================
    # Sidebar Layout
    # ====================

    def _build_sidebar(self):
```

This is a **private method** responsible for creating and arranging the left-hand sidebar. It's where all the settings-related widgets are placed.

📦 Sidebar Container

```
    """Creates the left sidebar for settings."""
    sidebar = ctk.CTkFrame(self, width=180, corner_radius=20,
fg_color=self.theme["frame_color"])
```

- `ctk.CTkFrame`: A styled frame from `customtkinter` used to hold sidebar content.
- width=180: Sets a fixed width for the sidebar.
- `corner_radius`=20: Gives the frame rounded corners for a modern UI look.
- `fg_color`: Uses the theme's frame color for background consistency.

```
    sidebar.grid(row=0, column=0, sticky="ns", padx=10, pady=10)
```

- Places the sidebar in **column 0** and **row 0** of the main window's grid layout.
- `sticky="ns"`: Ensures the sidebar stretches vertically (north to south).
- `padx` and `pady`: Adds spacing around the sidebar for better layout aesthetics.

```
    sidebar.grid_propagate(False)
```

- Prevents the frame from resizing to fit its children, enforcing the fixed width.

🏷️ Sidebar Header

```
    # Sidebar Header
    ctk.CTkLabel(sidebar, text="CmdMe", font=("Segoe UI", 26, "bold"),
            text_color=self.theme["text_color"]).pack(pady=(30, 15))
```

- Adds a large, bold header label with the app's name.
- The `pack()` layout manager is used here for vertical stacking with padding.

🗑️ Clear Chat Button

```
    # Clear Chat Button
    ctk.CTkButton(sidebar, text="🗑 Clear Chat",
fg_color=self.theme["button_fg"],
              hover_color=self.theme["button_hover"], corner_radius=10,
              command=self.clear_chat).pack(pady=(0, 20), fill='x',
padx=15)
```

- A button to clear the chat window, styled with theme colors.
- Uses an emoji (🗑) to visually indicate the function.
- `command=self.clear_chat`: Links the button to a method that clears the chat.
- `fill='x'`: Expands the button to fill the horizontal space.

⚙️ Settings Header

```
        # Settings Header
        ctk.CTkLabel(sidebar, text="⚙ Settings", font=("Segoe UI", 18,
"bold"),
                     text_color=self.theme["text_color"]).pack(pady=(0, 10))
```

- A smaller header under the chat button to visually group the upcoming settings controls.
- The gear emoji adds clarity.

🎛️ Settings Controls

```
        # Settings Controls
        self.setting_widgets = {}
```

- Initializes a dictionary to store references to each setting widget (dropdowns, sliders, etc.), allowing easy retrieval and manipulation later.

```
        self._create_dropdown(sidebar, "Appearance Mode",
"appearance_mode", ["light", "dark", "system"])
```

- Adds a dropdown menu to choose between light, dark, or system theme modes.

```
        self._create_slider(sidebar, "Voice Speed", "voice_rate", 100, 250)
```

- Adds a slider widget to control the speech rate of the voice assistant.

```
        self._create_dropdown(sidebar, "Voice Gender", "voice_gender",
["male", "female"])
```

- Another dropdown to switch between male and female voices.

These helper methods (`_create_dropdown` and `_create_slider`) are defined elsewhere in the class and handle the UI logic for creating these elements while storing them in `self.setting_widgets`.

✅ Apply & 🔁 Reset Buttons

```
        # Apply/Reset Buttons
        ctk.CTkButton(sidebar, text="☑ Apply",
command=self._apply_app_settings,
                      fg_color=self.theme["button_fg"],
hover_color=self.theme["button_hover"], corner_radius=10).pack(pady=(20,
5), fill='x', padx=15)
```

- The **Apply** button updates the application settings based on the user's changes.

```
    ctk.CTkButton(sidebar, text="🔁 Reset",
command=self._reset_app_settings,
              fg_color=self.theme["button_fg"],
hover_color=self.theme["button_hover"], corner_radius=10).pack(pady=(0,
10), fill='x', padx=15)
```

- The **Reset** button restores default settings (e.g., default voice, speed, and theme).

These controls make it easy for users to experiment with preferences and revert if needed.

✅ **Summary**

The `_build_sidebar()` method creates a clean, functional, and thematically consistent sidebar for CmdMe. This sidebar is the control hub for the user, offering essential customization options like theme, voice speed, and gender, along with handy actions like clearing the chat or resetting preferences.

9.10 Main Display Area: Chat and Command Interface

While the sidebar provides settings, the heart of the CmdMe application lies in the **main display area** — a space for conversation, command input, and user interaction. In this section, we break down how this part of the GUI is built and explore each component's role and behavior.

🧱 `_build_main()` **Method**

```
  # ====================
  # Main Display Area
  # ====================

  def _build_main(self):
     """Builds the main chat and input interface."""
```

This method is responsible for constructing the right-hand side of the window — the core interaction zone.

🖼️ **Main Frame**

```
    main_frame = ctk.CTkFrame(self, corner_radius=20,
fg_color=self.theme["frame_color"])
    main_frame.grid(row=0, column=1, sticky="nsew", padx=10,
pady=10)
```

- `main_frame`: A container that holds all components of the main interface.
- It's placed in **column 1** to sit beside the sidebar (`column 0`).
- sticky="`nsew`": Ensures the frame stretches in all directions.

👋 Greeting Header

```
      # Header
      ctk.CTkLabel(main_frame, text="Hello 👋, how can I assist you
today?",
                   font=("Segoe UI", 20, "bold"),
text_color=self.theme["text_color"]).pack(pady=(25, 15))
```

- A friendly greeting shown at the top of the chat interface.
- Sets the tone for a conversational assistant.

💬 Chat Frame

```
      # Chat Frame
      chat_frame = ctk.CTkFrame(main_frame, corner_radius=20,
fg_color=self.theme["widget_color"])
      chat_frame.pack(expand=True, fill='both', padx=20, pady=(5, 15))
```

- A frame that wraps around the scrolling chat box.
- Uses `expand=True` and `fill='both'` so it resizes dynamically with the window.

📝 Chat Display

```
      self.chat_display = scrolledtext.ScrolledText(
         chat_frame, wrap='word', font=("Segoe UI", 12),
         bg=self.theme["chat_bg"], fg=self.theme["chat_fg"],
         relief='flat', bd=0, insertbackground=self.theme["chat_fg"]
      )
```

- `ScrolledText`: A built-in `tkinter` widget that allows scrolling through conversation history.
- `wrap='word'`: Ensures text wraps nicely at word boundaries.
- Background and foreground colors follow the current theme.

```
      self.chat_display.pack(expand=True, fill='both', padx=12,
pady=12)
      self.chat_display.insert("end", "Assistant: Welcome! Type or
speak a command below.\n\n")
```

- The chat is initialized with a welcome message.
- `insert("end", ...)` adds text to the bottom of the chat.

⌨️ Input Section

```
      # Input Section
      input_frame = ctk.CTkFrame(main_frame, height=60,
corner_radius=20, fg_color=self.theme["widget_color"])
      input_frame.pack(fill='x', padx=20, pady=(0, 20))
```

- A dedicated frame for command input widgets — the text entry and buttons.
- Positioned at the bottom of the main frame.

🔡 Command Entry Field

```
    self.entry = ctk.CTkEntry(input_frame, placeholder_text="Type
your command here...",
                              font=("Segoe UI", 12), corner_radius=10)
```

- The entry field where users can type commands.
- `placeholder_text` guides the user on what to do.

```
    self.entry.pack(side='left', fill='x', expand=True, padx=(10,
10), pady=10)
    self.entry.bind("<Return>", lambda e: self._handle_send())
```

- `bind("<Return>")`: Pressing **Enter** triggers the command send function.
- Entry expands to take up remaining horizontal space.

📩 Send & 🎤 Speak Buttons

```
    # Send & Speak Buttons
    ctk.CTkButton(input_frame, text="📩 Send",
fg_color=self.theme["button_fg"],
                  hover_color=self.theme["button_hover"],
corner_radius=10,
                  command=self._handle_send).pack(side='left', padx=(0,
10))
```

- A button to submit the typed command.

```
    ctk.CTkButton(input_frame, text="🎤 Speak",
fg_color=self.theme["button_fg"],
                  hover_color=self.theme["button_hover"],
corner_radius=10,
                  command=self._handle_speak).pack(side='left')
```

- A button to activate **voice input** mode.

Together, these buttons offer two input modes: **text** and **voice**, giving users flexibility in how they interact.

➢ **Summary:**

The `_build_main()` function builds the primary interface of the CmdMe application — a responsive chat area, input field, and dual interaction buttons. It emphasizes ease of communication, whether users prefer typing or speaking.

9.11 Settings Controls: Personalizing the Assistant

The user experience is enhanced when personalization is possible. The CmdMe application gives users the ability to adjust **appearance**, **voice speed**, and **voice gender**. This is done through intuitive dropdowns and sliders on the sidebar. In this section, we'll go through how each control is implemented and works behind the scenes.

🎛️ `_create_dropdown()` – Creating Dropdown Menus

```
    # ====================
    # Settings Controls
    # ====================

    def _create_dropdown(self, parent, label, key, options):
        ctk.CTkLabel(parent, text=label,
text_color=self.theme["text_color"]).pack(pady=(10, 0))
        dropdown = ctk.CTkOptionMenu(parent, values=options, command=lambda
val: self._update_setting(key, val))
        dropdown.set(self.settings[key])
        dropdown.pack(pady=5, fill='x', padx=10)
        self.setting_widgets[key] = dropdown
```

Purpose: Adds a labeled dropdown to the given `parent` widget (in this case, the sidebar).

- **Label**: A simple text label is added above the dropdown to describe the setting (e.g., *Appearance Mode*).
- **CTkOptionMenu**: A themed dropdown menu from `customtkinter`.
- **Command Binding**: When the user selects a new value, `_update_setting()` is called to update internal settings.
- **Initialization**: The dropdown is set to the current value from the settings dictionary.
- **Storage**: Each dropdown is stored in `self.setting_widgets` for later reset.

🎚️ `_create_slider()` – Creating Sliders for Numeric Settings

```
    def _create_slider(self, parent, label, key, min_val, max_val):
        ctk.CTkLabel(parent, text=label,
text_color=self.theme["text_color"]).pack(pady=(10, 0))
        slider = ctk.CTkSlider(parent, from_=min_val, to=max_val,
number_of_steps=15,
                               command=lambda val: self._update_setting(key,
int(val)))
        slider.set(self.settings[key])
        slider.pack(pady=5, fill='x', padx=10)
        self.setting_widgets[key] = slider
```

Purpose: Creates a slider that lets users set values like **voice speed**.

- `from_` and `to`: Defines the slider range.
- `number_of_steps`: Determines slider precision.
- `command`: When moved, the slider updates the associated setting.
- Again, the current setting is loaded initially and stored for reset purposes.

⚙️ `_update_setting()` – Dynamic Setting Updates

```
    def _update_setting(self, key, value):
        self.settings[key] = value
```

A very simple method: it updates the internal `settings` dictionary when a control changes.

✅ `_apply_app_settings()` – Apply the New Settings

```
    def _apply_app_settings(self):
        ctk.set_appearance_mode(self.settings["appearance_mode"])
        self._apply_theme()
        self._refresh_colors()
        update_voice_settings(self.settings["voice_rate"],
self.settings["voice_gender"])
        speak("Settings applied.")
```

This method is triggered when the **Apply** button is clicked.

- **Appearance Mode**: Updates light/dark mode.
- **Theme Refresh**: Applies the color scheme again using `_apply_theme()` and `_refresh_colors()`.
- **Voice Settings**: Passes rate and gender to the voice engine.
- Gives audible confirmation using `speak()`.

🔄 `_reset_app_settings()` – Reset to Default

```
    def _reset_app_settings(self):
        self.settings = {"appearance_mode": "system", "voice_rate": 150,
"voice_gender": "female"}
        for key, widget in self.setting_widgets.items():
            if isinstance(widget, ctk.CTkOptionMenu):
                widget.set(self.settings[key])
            elif isinstance(widget, ctk.CTkSlider):
                widget.set(self.settings[key])
        self._apply_app_settings()
        speak("Settings reset to default.")
```

- Restores default settings.
- Loops through the widgets and updates their values using `.set()`.
- Reapplies the settings and confirms via speech.

🎨 `_refresh_colors()` – Live Theme Switching

```
    def _refresh_colors(self):
```

```
        self.chat_display.config(bg=self.theme["chat_bg"],
fg=self.theme["chat_fg"],
                                 insertbackground=self.theme["chat_fg"])
        self.configure(fg_color=self.theme["bg_color"])
```

- Updates chat display and app background to match the new theme settings.
- This method ensures immediate visual feedback when switching between dark and light modes.

➢ **Summary:**

This section empowers users to **personalize their assistant** easily:

- Dropdowns adjust modes and voice preferences.
- Sliders fine-tune voice speed.
- Apply/Reset buttons make these changes instant and reversible.

With this structure, CmdMe feels modern and user-centric — a smart assistant that can adapt to individual preferences.

9.12 Input Handlers: Capturing and Processing User Intent

At the core of CmdMe's interactivity lies its ability to understand **what the user wants** — whether typed manually or spoken aloud. This final GUI layer completes the assistant's loop of interaction by defining three essential methods:

- **Clearing the chat**
- **Sending typed commands**
- **Capturing voice commands**

🧹 `clear_chat()` – **Starting Fresh**

```
    # ====================
    # Input Handlers
    # ====================

    def clear_chat(self):
        self.chat_display.delete("1.0", "end")
```

Functionality: Removes all existing chat content from the interface.

- "`1.0`" refers to the first character in the text box (line 1, char 0).
- "`end`" deletes everything till the end of the widget.

- This is bound to the "🗑️ Clear Chat" button in the sidebar.

This allows users to declutter the UI and start a fresh conversation, especially useful in long sessions.

📤 `_handle_send()` – Submitting Text Input

```
def _handle_send(self):
    text = self.entry.get().strip()
    if text:
        self.chat_display.insert("end", f"You: {text}\n")
        self.chat_display.see("end")
        self.entry.delete(0, 'end')
        process_command(text, self.chat_display)
```

This method handles **typed input** in the entry field.

1. `self.entry.get().strip()`: Reads the user's typed input and removes leading/trailing whitespace.
2. If there's text, it:
    - Displays it in the chat window with "`You: ...`" prefix.
    - Automatically scrolls to the latest message using `.see("end")`.
    - Clears the input field `(.delete(0, 'end'))`.
    - Sends the command to `process_command()` for interpretation and action.

This ensures that typed input is treated exactly the same way as spoken commands — allowing seamless switching between modalities.

🎙️ `_handle_speak()` – Activating Voice Listening

```
def _handle_speak(self):
    self.chat_display.insert("end", "Listening for voice command...\n")
    self.chat_display.see("end")
    listen_voice(self.chat_display)
```

Voice interaction entry point. When the "🎤 Speak" button is clicked:

- It notifies the user in the chat window that voice input is being listened for.
- Calls the previously defined `listen_voice()` function (see Chapter 9.6), which:
    - Activates the microphone
    - Uses Google's speech recognition to capture and convert voice to text
    - Sends the resulting command to `process_command()`

This bridges the voice recognition pipeline into the visual GUI, giving a cohesive feel to the application.

🧠 Integration Summary

These three methods complete the **input-processing-output loop** of the assistant:

Action	Method	Output
Clear chat	`clear_chat()`	Empties the chat window
Type + Enter	`_handle_send()`	Runs typed command
Click 🎤 Speak	`_handle_speak()`	Runs voice input pipeline

By abstracting these input handlers, the assistant keeps its interface **clean**, **modular**, and **user-friendly**.

9.13 Application Entry Point: Launching the Assistant

At the very end of our source code lies a small but **crucial block** of logic — the one that actually **starts** our assistant application:

```
# ================================
# Application Entry Point
# ================================

if __name__ == "__main__":
   app = AssistantApp()
   app.mainloop()
```

🧠 **What does this do?**

This conditional block acts as a **program gatekeeper**. Let's understand it line by line.

✅ `if __name__ == "__main__":`

This is a **standard Python idiom** that checks whether the file is being run **directly** (not imported as a module in another script).

- If this condition is `True`, it means: "Run this script now."
- If the script is **imported elsewhere**, this block is **ignored**, preventing unintended execution.

💡 Think of this like a "main gate" — it ensures your app runs only when you **explicitly** execute the script.

🧱 `app = AssistantApp()`

Here, the main GUI class (`AssistantApp`, defined earlier) is **instantiated**.

- All the GUI components (sidebar, chat box, buttons, themes) are initialized.
- Settings are loaded.

- Voice engine setup is triggered.
- The app window is configured and made ready to use.

This single line constructs everything you've built in the previous chapters into a **live, interactive window**.

🔁 `app.mainloop()`

This tells `Tkinter` (via `customtkinter`) to **enter the event loop**:

- It keeps the window open.
- Listens for clicks, typing, and voice input.
- Handles all user interactions **asynchronously**.

Once `.mainloop()` is called, the assistant is up and running, ready to serve the user until the window is closed.

🧩 Why Is This Important?

This structure provides **clean code organization**. It:

- Keeps your code modular and import-safe.
- Separates logic from execution.
- Follows Python best practices, especially for larger or multi-file projects.

✅ Summary

Line	Purpose
`if __name__ == "__main__":`	Ensures this script runs standalone
`app = AssistantApp()`	Instantiates the full GUI assistant
`app.mainloop()`	Enters Tkinter's event loop

This final section activates your assistant and brings everything to life. 🎉

In the next chapter, we will **package this application into a standalone executable file (.exe)** — allowing users to run it **without installing Python** or any dependencies.

10 Packaging

In this chapter, we will learn how to **package our Python voice assistant application into an executable file (.exe)** that can run independently on any Windows machine without requiring Python to be installed.

10.1 Overview

Python applications are interpreted, which means users typically need Python installed to run them. However, using tools like **PyInstaller**, we can bundle all required files (Python runtime, libraries, assets) into a single `.exe` file.

This process involves:

- Installing the `PyInstaller` library.
- Preparing necessary files and folders.
- Writing a build command.
- Testing the output `.exe`.

10.2 Installing PyInstaller

Before we start, ensure that your Python environment is active. Then, install PyInstaller by running the following command in your terminal or command prompt:

```
pip install pyinstaller
```

This will download and install the tool needed to convert your Python script into an executable file.

10.3 Structuring Your Project Directory

To avoid packaging issues, your project folder should be organized. For example:

```
CmdMe/
├── main.py
├── icon.ico
├── requirements.txt
└── assets/ (if any)
```

Make sure all files (like icon.ico) are accessible in your code using relative paths.

10.4 Writing the Packaging Command

To build the `.exe` file, use the following command:

```
pyinstaller --onefile --windowed --icon=icon.ico main.py
```

Explanation of flags:

- `--onefile`: Combines everything into a single `.exe`.
- `--windowed`: Prevents the console window from showing up (important for GUI apps).
- `--icon=icon.ico`: Adds a custom application icon.

This will generate a `dist/` folder containing your `.exe` file.

10.5 Testing the Executable

1. Navigate to the `dist/` directory.
2. Double-click the `.exe` file.
3. The assistant should launch with the expected GUI and features.
4. Ensure all resources like the icon and voice features work correctly.

10.6 Common Issues and Fixes

Issue	Cause	Solution
Missing modules	Not found by PyInstaller	Add a `--hidden-import` argument
File not found	Improper path	Use `sys._MEIPASS` logic in your code
Voice not working	TTS engine missing	Ensure `pyttsx3` dependencies are bundled

10.7 Conclusion

By the end of this chapter, your Python assistant application is now a **fully packaged `.exe` file**, ready to be shared with others — **no Python installation required**. This step is crucial for distribution, deployment, or demonstration purposes in both academic and professional contexts.

In the next chapter, we will discuss **final testing, versioning, and optional enhancements** such as auto-updates, tray minimization, or packaging into an installer.

11 Installer

Now that we have packaged our Python application into a single executable using PyInstaller, the next step is to create a professional installer. This allows end-users to install the app on their system like any other software — with a proper setup wizard, start menu shortcut, and uninstall options.

We'll use a free and widely used tool: **Inno Setup**.

11.1 What is Inno Setup?

Inno Setup is a Windows installer authoring tool. It lets you create .exe installer files that can:

- Show a setup wizard
- Install your `.exe` file to `Program Files`
- Add a desktop/start menu shortcut
- Allow users to uninstall the app from Control Panel

11.2 Download and Install Inno Setup

1. Visit: https://jrsoftware.org/isinfo.php
2. Download the latest version of Inno Setup.
3. Install it on your system.

11.3 Basic Inno Setup Script

Once installed, create a new script with the following minimal configuration:

```
[Setup]
AppName=CmdMe Assistant
AppVersion=1.0
DefaultDirName={pf}\CmdMe
DefaultGroupName=CmdMe
OutputDir=dist-installer
OutputBaseFilename=CmdMeSetup
Compression=lzma
SolidCompression=yes
SetupIconFile=icon.ico

[Files]
```

```
Source: "dist\main.exe"; DestDir: "{app}"; Flags: ignoreversion

[Icons]
Name: "{group}\CmdMe Assistant"; Filename: "{app}\main.exe"
Name: "{commondesktop}\CmdMe Assistant"; Filename: "{app}\main.exe";
Tasks: desktopicon

[Tasks]
Name: "desktopicon"; Description: "Create a &desktop icon";
GroupDescription: "Additional icons:"
```

💡 Customize paths like main.exe, icon.ico and OutputDir based on your actual file structure.

11.4 Building the Installer

1. Open Inno Setup Compiler.
2. Paste your script or use the wizard.
3. Press **Build** or use F9.
4. The .exe setup installer will be generated in your output directory.

11.5 Testing the Installer

- Run the installer on a different machine (without Python).
- Confirm that:
    - Setup wizard appears.
    - Files are installed to `Program Files\CmdMe`.
    - Shortcut is added to Desktop or Start Menu.
    - Application launches as expected.
    - App is listed in "Add or Remove Programs".

11.6 Conclusion

With the successful creation of an installer using Inno Setup, the voice assistant application has reached a stage where it can be distributed and installed on end-user systems in a standardized and professional manner. The packaging process ensures that all necessary components are bundled and deployed seamlessly, enabling ease of access and usability for non-technical users.

12 References

This section lists all the resources, libraries, documentation, and tools referred to or utilized in the development of the *CmdMe Voice Assistant* project. Proper attribution ensures academic integrity and provides readers with sources for further learning or verification.

12.1. Python Libraries and Frameworks

- Tom Schimansky. **`CustomTkinter`***: Modern UI for* **`Tkinter`**. GitHub Repository. Available at: https://github.com/TomSchimansky/CustomTkinter
- Pyttsx3 Developers. **`pyttsx3`***: Text-to-speech conversion library in Python*. Documentation. Available at: https://pyttsx3.readthedocs.io/
- Uberi. **`SpeechRecognition`***: Library for performing speech recognition, with support for several engines and APIs*. PyPI. Available at: https://pypi.org/project/SpeechRecognition/
- Python Official Docs. **`Subprocess`** *module documentation*. Available at: https://docs.python.org/3/library/subprocess.html
- Python Official Docs. **`Webbrowser`** *module documentation*. Available at: https://docs.python.org/3/library/webbrowser.html
- Python Official Docs. **`Datetime`** *module documentation*. Available at: https://docs.python.org/3/library/datetime.html
- Python Official Docs. **`OS`** *module documentation*. Available at: https://docs.python.org/3/library/os.html
- Python Official Docs. *Threading module documentation*. Available at: https://docs.python.org/3/library/threading.html

12.2. Design and Documentation Tools

- Microsoft Word. *Used for project documentation and report writing.*
- Visual Studio Code. *Used for writing and debugging Python code.* Available at: https://code.visualstudio.com/

Note: All resources and libraries mentioned are open-source and were used in compliance with their respective licenses for educational purposes.

13 Conclusion

The development of **CmdMe**, a voice-activated personal assistant, has been a rewarding journey combining multiple aspects of software development—from GUI design and speech technologies to system integration and multithreading. The project successfully demonstrates how Python can be used to create intelligent, user-friendly desktop applications that provide both convenience and customization to the end user.

Through the implementation of `customtkinter`, we created a visually appealing and responsive user interface that supports both light and dark themes, aligning with modern UI standards. Integration with `pyttsx3` and `speech_recognition` allowed the assistant to communicate naturally, bridging the gap between human interaction and machine execution.

One of the most impactful features of CmdMe is its ability to process both voice and text input, making it versatile and accessible. The modular design of command handlers ensures easy scalability, allowing future enhancements such as integration with APIs, databases, or AI models.

This project not only reinforced foundational programming skills but also introduced real-world problem-solving scenarios involving system commands, voice I/O, and multithreading. The ability to configure settings such as voice gender, speaking rate, and appearance mode reflects an emphasis on user personalization—a critical factor in today's software landscape.

In conclusion, It is a practical demonstration of how user experience, system interaction, and programming logic can converge to build something intelligent and functional. While the current version of CmdMe offers essential assistant capabilities, it lays the groundwork for more advanced features in the future. With further enhancements, this assistant has the potential to evolve into a more powerful and integrated productivity tool.