Feature prompts

Text-Based Query Handling (OpenAI Integration)

Goal: I need a Python script that loads an API key from a `.env` file using `python-dotenv`, then sends a simple prompt to the OpenAI API and prints the response.

Requirements:

1. The script should include debug prints verifying the `.env` file is loaded.
2. It should handle errors gracefully and print any exceptions.
3. It should use `gpt-3.5-turbo` or `text-davinci-003` for the OpenAI API.
4. The script must be runnable inside a standard virtual environment on Windows using Python 3.x.

Output:

- Provide a code snippet that fully demonstrates how to load `.env` variables, set `openai.api_key`, send a prompt, and print the AI's response.
- Include inline comments or a short section explaining each major step.
- Show me how to test the script via the terminal in Windows (with the virtual environment activated).

Goal: Extend my existing Python-based assistant project, "Kyle." I have already set up the OpenAI integration, and now I want to add additional features. Specifically:

1. **Notion Integration** for note-taking:
    - Authenticate with Notion's API (using a secret token).
    - Create functions to add, read, and update notes or tasks within a specified Notion database.
2. **Reminder System**:

- Add a function to schedule reminders (in Python) that notifies me at a certain time or after a specified delay.
- Provide at least one example approach for scheduling tasks (e.g., using `schedule` library, or another lightweight approach).

3. **Voice Interaction**:
   - Integrate speech-to-text using `speech_recognition`.
   - Integrate text-to-speech (e.g., `pyttsx3`).
   - Demonstrate how the user can speak a command (e.g., “Take a note” or “Set a reminder...”) and have the assistant respond audibly.

Requirements:

- Provide a code snippet in Python showing how to:
  1. Set up the Notion API client.
  2. Create a new note in a Notion database by voice command.
  3. Set a simple reminder in code.
  4. Use speech-to-text for commands and text-to-speech for responses.
- Include relevant inline comments explaining each major step.
- Show me how to install and set up any additional required libraries.
- Assume I'm on Windows using a virtual environment.

Output:

- A single, consolidated Python script (or multiple short snippets) that demonstrates these features.
- Briefly describe how to run and test each feature from the terminal.

You are ChatGPT, an expert AI assistant. I’m building my own AI assistant named “Kyle” using Python on a Windows machine. Here’s what I want:

1. **Core Features:**

- **OpenAI Integration**: Kyle should handle text-based queries, chat with me using the OpenAI API (e.g., `gpt-3.5-turbo`).
- **Voice Interaction**:
  - Speech-to-text with `speech_recognition`.
  - Text-to-speech with `pyttsx3` or another TTS library.
- **Note-Taking & Reminders**:
  - Integrate with Notion's API for creating and reading notes.
  - Provide a simple reminder system (e.g., using `schedule` or another library).
- **Context Awareness**: Kyle should remember recent instructions in a session (short-term memory) and store some user preferences or frequently used data in a local database (long-term memory).
- **Personality**: Allow me to define a specific tone or style for Kyle's responses.

2. **Implementation Requirements**:
   - Use a **virtual environment** on Windows.
   - Provide **step-by-step** instructions on installing and importing required libraries (e.g., `openai`, `speechrecognition`, `pyttsx3`, `notion-client`, `schedule`).
   - Outline or provide a **Python script** (or multiple scripts) that demonstrate these features.
   - Include **comments** or docstrings explaining critical parts of the code.
   - Show me **how** to run and test each feature in the command line (including how to set environment variables like `OPENAI_API_KEY`).
3. **Project Setup & Structure**:
   - Recommend a clean project structure (e.g., one folder with `env`, `main.py`, `notion_integration.py`, etc.).
   - Show me how to **modularize** Kyle's functionalities (voice, Notion integration, reminders, etc.) so the code remains maintainable.

4. **Output**:
    - Provide **example code** with inline explanations.
    - If possible, show me how to make Kyle greet me, listen for a command, respond verbally, and optionally create a Notion note or set a reminder.
    - Offer **best practices** for error handling, storing secrets (API keys), and potentially upgrading Kyle with more advanced capabilities later (like browsing or advanced context memory).

Based on this, please generate a comprehensive guide and sample code to help me build Kyle from scratch.

Long-Term Memory (Basic Version)

Goal: My Python-based AI assistant, "Kyle," needs **long-term memory** so it can remember and reference past information (e.g., user preferences, conversation history). I'm on a Windows machine, using a virtual environment.

Requirements:

1. **Basic Approach (SQLite or JSON)**:
    - Demonstrate how to store and retrieve text data (user input, assistant's replies, or random facts) in a local database or file.
    - Provide a **keyword-based** retrieval example (search by matching words).
    - Include code samples showing table creation, insertion, and retrieval.
2. **Advanced Approach (Vector Embeddings)**:
    - Use **sentence-transformers** (e.g., `all-MiniLM-L6-v2`) or **OpenAI Embeddings**.
    - Show how to store embeddings in a local vector database (e.g., **FAISS**) or a managed service.
    - Demonstrate **semantic** recall (i.e., retrieving information by meaning rather than exact keywords).
    - Provide code snippets for both storing and querying embeddings.
3. **Implementation Details**:

- Explain how to integrate these memory calls into the assistant's conversation flow (i.e., store new info after each user input, retrieve relevant items before generating a response).
- Include **prompt engineering** best practices for adding retrieved memory into GPT's context.
- Offer any **performance** or **scalability** considerations for large datasets.

4. **Optional**:
    - Summarize or prune older memory to keep the DB and vector index manageable.
    - Provide tips for **error handling** and **logging** (so I can see what's being recalled).

Output:

- A **step-by-step guide** and **complete code snippets** implementing both the basic (SQLite) and advanced (embeddings) approaches, with explanations on how to run and test each.
- Recommendations on which approach might fit best depending on my scale and complexity needs.

Goal: Add **advanced long-term memory** to my Python-based assistant, "Kyle," using **vector-based semantic retrieval**. Kyle should be able to store and retrieve information based on meaning, not just keywords, allowing for more natural and context-aware responses.

Requirements:

1. **Embedding Model**:
    - Use an **embedding model** (e.g., `sentence-transformers` with `all-MiniLM-L6-v2`, or OpenAI's `text-embedding-ada-002`).
    - The model should convert text into vector representations for semantic comparison.

2. **Vector Database**:
    - Implement a vector database for storing and querying embeddings:
        - Use **FAISS** (local storage) or **Pinecone/Weaviate** (cloud options).
    - Add helper functions:
        - `store_memory(content: str)`: Store the embedded representation of a piece of text.
        - `retrieve_memory(query: str, top_k=3) -> list[str]`: Retrieve top `k` semantically similar pieces of memory.
3. **Integration with the Assistant**:
    - When processing a user query, embed the query text and retrieve relevant memories to add context to the response.
    - Example use case: If the user previously mentioned liking sci-fi books, and later asks, "Any recommendations?" Kyle should recall that preference.
4. **Performance and Optimization**:
    - Ensure the memory store is efficient and doesn't slow down over time.
    - Add a memory pruning/summarization function to condense older entries when the dataset grows large.
5. **Error Handling and Security**:
    - Handle cases where the database is empty or where the query returns no relevant results.
    - Securely store data and avoid exposing sensitive information.

Output:

- A **step-by-step guide** with code demonstrating:
    1. Installing necessary libraries (`faiss`, `sentence-transformers`, etc.).
    2. Initializing and using the embedding model.
    3. Setting up and using the vector database (FAISS or cloud-based).
- Provide example Python functions:

```
python
Copy code
store_memory("User likes sci-fi books.")
results = retrieve_memory("What kind of books does the user like?")
```

- Show how to integrate the retrieved memories into the OpenAI API prompt to enhance context-aware conversation.
- Include recommendations for:
  - Reducing memory bloat (e.g., summarizing older conversations).
  - Handling rate limits (if using cloud services).

Optional Enhancements:

- Allow the user to **label** certain memories (e.g., "important," "personal") for easier querying.
- Support for **multi-session memory** so different conversation threads can have separate memory contexts.

Web Browsing and Real-Time Information Retrieval

Goal: Add **web browsing** and **real-time information retrieval** capabilities to my Python-based assistant, "Kyle." I want Kyle to be able to perform general web searches and fetch specific types of real-time data (e.g., weather, stock prices, news summaries).

Requirements:

1. **Web Search**:
   - Integrate a web search API (e.g., **SerpAPI**, **Bing Web Search API**, or an open-source alternative).
   - Provide functions like `search_web(query: str)` that return summarized results (top headlines, links, etc.).
   - If possible, enable filtering (e.g., "only return news articles").

2. **Real-Time Data Retrieval**:
    - Weather: Retrieve weather data for a specified location using an API (e.g., OpenWeatherMap, WeatherAPI).
    - Stock Prices: Fetch the latest stock prices for specified symbols.
    - News Summaries: Retrieve recent news articles related to specific topics.
3. **Implementation Details**:
    - Use Python libraries (`requests`, `json`, etc.) for API calls.
    - Create helper functions:
        - `get_weather(location: str) → str`
        - `get_stock_price(symbol: str) → str`
        - `get_news_summary(topic: str) → list[str]`
    - Include API key configuration (via `.env` file or environment variables).
    - Ensure results are formatted cleanly for both text and voice responses (e.g., trimming excess data).
4. **Optional Enhancements**:
    - Add a fallback function using Python's built-in `webbrowser.open()` to open a search result in the user's browser.
    - Add a caching mechanism to store recent queries and avoid redundant API calls.

Output:

- A **step-by-step guide** showing how to:
    1. Install and configure the chosen APIs.
    2. Add API keys securely using `.env` files.
    3. Test web searches and real-time data functions with example queries.
- Provide **Python code snippets** demonstrating the above functionality and showing how to integrate it with the main assistant flow.

- Offer **recommendations** for API rate limits, error handling, and fallback strategies (e.g., if API calls fail).

Context-Aware Conversations

Goal: Add **context-aware conversation** capabilities to my Python-based assistant, “Kyle.” The assistant should be able to retain short-term context during conversations, allowing it to reference previous user inputs and respond coherently across multi-step interactions.

Requirements:

1. **Conversation History Tracking**:
    - Implement a system to track and store the last few interactions (user queries and assistant responses).
    - Add a configurable **history window size** (e.g., track the last `n` interactions).
2. **Memory Integration**:
    - Combine short-term memory (conversation history) with **long-term memory retrieval**.
    - The assistant should prioritize short-term memory for immediate interactions but reference long-term memory when necessary (e.g., "Remind me what we talked about yesterday.").
3. **Handling Multi-Turn Conversations**:
    - Allow Kyle to handle **follow-up questions** by recognizing implicit context (e.g., "What's the weather like today?" → "And tomorrow?").
    - Maintain references to pronouns, topics, or incomplete follow-ups (e.g., "Tell me more about that.").
4. **Prompt Engineering**:
    - Construct the prompt for the OpenAI API to include relevant past interactions in a structured way:

- Example format:

```
vbnet
Copy code
Assistant Memory:
- User: What's the best sci-fi book?
- Assistant: I recommend "Dune" by Frank Herbert.
- User: Who’s the author again?
```

- Provide a function to **dynamically trim the history** if the prompt size exceeds the model's token limit.

5. **Handling Edge Cases**:
   - Handle when there is **no context available** (e.g., first interaction in a session).
   - Detect when the conversation context changes and clear/reset history if needed (e.g., “Forget what we were discussing”).

Implementation Details:

- Create helper functions:
  - `add_to_conversation_history(user_input: str, assistant_response: str)`: Adds a new interaction to the history.
  - `generate_contextual_prompt(current_query: str) → str`: Constructs the OpenAI API prompt by combining relevant conversation history and the current query.
- Store history in-memory (list or dictionary) for quick reference.

Output:

- A **step-by-step guide** showing how to:
  1. Track and store conversation history.
  2. Construct contextual prompts dynamically and handle long prompts by trimming.
  3. Reset or prune the conversation history when necessary.

- Provide **Python code snippets** implementing these functions.
- Explain how to handle large prompts without exceeding the OpenAI token limit.

Optional Enhancements:

- Add a **reset command** (“Let’s start over”) to clear the conversation history.
- Include a way for Kyle to **ask for clarification** if context is ambiguous (e.g., "Did you mean X or Y?").
- Implement a **summary function** that periodically summarizes previous interactions to reduce token usage while maintaining key details.

Improved Speech Recognition for Accents

Goal: Significantly improve the speech recognition accuracy for my AI assistant, especially for my accent. I’m currently using Python on Windows, and I want to explore options beyond the default SpeechRecognition library.

Requirements:

1. Suggest and compare **different speech-to-text engines** (Google Cloud Speech-to-Text, Azure, Amazon Transcribe, Vosk, Coqui STT) in terms of accuracy, cost, and ease of setup.
2. Show **how to implement** at least one cloud-based solution (e.g., Google Cloud) with custom language codes or phrase hints for better accent recognition.
3. Demonstrate **basic offline** approaches (Vosk or Coqui STT) that I can fine-tune for my accent, including how to set up or train these engines if possible.
4. Provide **practical tips** for better results (microphone choice, environment noise, adjusting `pause_threshold` in SpeechRecognition, etc.).
5. Summarize **best practices** for integrating any chosen engine into an existing Python assistant, including code samples.

Output:

- A short, **step-by-step guide** with **code snippets** demonstrating how to install, configure, and integrate the recommended engine(s).
- **Recommendations** for further tuning (e.g., custom vocabulary, wake words).
- Advice on **error handling** and **logging** to measure improvements in recognition accuracy.

Goal: Improve the speech recognition capabilities of my Python-based assistant, "Kyle," to better handle my accent and improve accuracy during voice interactions. I want to replace or enhance the current `speech_recognition` library with a more robust solution that supports diverse accents and custom vocabulary.

Requirements:

1. **Recognition Engine Options**:
    - Evaluate and implement a speech recognition engine that supports diverse accents and custom phrases:
        - **Local options**: `Vosk`, `Coqui STT` (formerly Mozilla DeepSpeech).
        - **Cloud options**: Google Cloud Speech-to-Text, Azure Cognitive Services, or Amazon Transcribe.
2. **Accent and Vocabulary Customization**:
    - Provide the ability to set the language/accent (e.g., `en-GB`, `en-ZA`, etc.) to better match my speaking style.
    - Add support for **custom phrases** and vocabulary hints to improve recognition for common words or phrases in my queries (e.g., "Kyle", "sci-fi books", "set a reminder").
3. **Implementation Details**:
    - Replace or complement `speech_recognition.Recognizer` with the chosen engine.
    - Show how to:
        - Install and configure the chosen engine.
        - Handle API keys securely (if using a cloud-based engine).

    - Capture and process the recognized text within the assistant's workflow.

4. **Performance Considerations**:
    - Ensure that the speech-to-text process runs in real-time or near real-time with minimal latency.
    - Handle network issues gracefully if using a cloud-based API (e.g., provide a fallback mode or informative error messages).

Output:

- A **step-by-step guide** for:
    1. Installing and configuring the speech recognition engine.
    2. Setting up language/accent options and custom phrases.
    3. Testing the speech recognition with sample phrases.
- Provide **Python code snippets** demonstrating:

```
python
Copy code
def recognize_speech_vosk() -> str:
    # Speech-to-text using Vosk
    ...

def recognize_speech_google() -> str:
    # Speech-to-text using Google Cloud API
    ...
```

- Recommendations for selecting the best engine (based on accuracy, speed, and cost).

Optional Enhancements:

- Include a **hotword detection** feature ("Hey Kyle") using libraries like `snowboy` or `porcupine`.

- Add the ability to **switch engines dynamically** (e.g., fallback to local recognition when offline).
- Implement **confidence thresholds** and user feedback to handle misrecognition (“Did you mean X?”).

Improved Logging and Debugging

Goal: Add a robust **logging and debugging** system to my Python-based assistant, "Kyle." The assistant should log all interactions, system-level actions, and errors, providing visibility into what it does at any given time and helping troubleshoot issues during development.

Requirements:

1. **Basic Logging**:
    - Log key events, including:
        - User inputs (text or voice).
        - Assistant responses.
        - System-level actions (e.g., “Opened Notepad,” “Set a reminder”).
        - Errors, exceptions, and API failures (with timestamps).
    - Format the logs for readability (e.g., timestamps, event types, etc.).
2. **Error Handling and Debugging**:
    - Add **try/except blocks** around key functions (e.g., API calls, file operations).
    - Log full stack traces for errors during development, but display user-friendly error messages during runtime.
3. **Log File Management**:
    - Store logs in a dedicated folder (e.g., `logs/`) with auto-rotating log files (to prevent large log files).
    - Implement a system to archive old logs and delete outdated ones.

4. **Debug Mode**:
    - Add a **debug mode** toggle that, when enabled:
        - Prints detailed logs to the console in addition to writing to a file.
        - Adds more verbose logs (e.g., function entry/exit points).
5. **Implementation Details**:
    - Use Python's `logging` module to set up:
        - `INFO`, `ERROR`, and `DEBUG` log levels.
        - Log to both a file and the console.
    - Create helper functions:

```
python
Copy code
def log_info(message: str):
    # Log information-level events
    ...

def log_error(message: str, exception: Exception):
    # Log errors with exception details
    ...
```

6. **User Query History**:
    - Allow the user to view recent interactions (from logs) by typing or asking, “What did I ask you earlier?”.

Output:

- A **step-by-step guide** for:
    1. Setting up the `logging` module with multiple handlers (file and console).
    2. Implementing log rotation and old log cleanup.
    3. Adding logging to key functions (e.g., API calls, voice commands, system commands).

- Provide **Python code snippets**:

```
python
Copy code
import logging
from logging.handlers import RotatingFileHandler

# Logging setup
logging.basicConfig(level=logging.INFO)
handler = RotatingFileHandler("logs/kyle.log", maxBytes=5 * 1024 * 1024,
backupCount=5)
logger = logging.getLogger(__name__)
logger.addHandler(handler)

logger.info("Assistant started")
try:
    # Simulate action
    1 / 0
except Exception as e:
    logger.error("Error during execution", exc_info=True)
```

- Show how to create a “debug mode” toggle that controls logging verbosity.

Optional Enhancements:

- Implement a **real-time log viewer** in the terminal or GUI (show logs as they happen).
- Add **filtering** to the log viewer (e.g., show only errors or interactions with a specific user).
- Allow the assistant to **summarize logs** upon request (e.g., “Summarize today’s activity”).

Personality Customization

Goal: Add **personality customization** to my Python-based assistant, "Kyle." I want to be able to switch between different personality modes (e.g., professional, casual, humorous) so that Kyle's responses are tailored based on the selected tone and style.

Requirements:

1. **Personality Modes**:
    - Create predefined modes (e.g., "Professional", "Friendly", "Humorous") with distinct response styles.
    - Add a command to switch modes dynamically during a conversation (e.g., "Kyle, be more formal," or "Switch to casual mode").
2. **Personality Configuration**:
    - Store personality settings (e.g., tone, formatting style) in a **configuration file** (`personality.json`) for easy updates.
    - Example JSON format:

```
json
Copy code
{
  "Professional": {
    "greeting": "Good day. How may I assist you?",
    "tone": "formal"
  },
  "Friendly": {
    "greeting": "Hey there! How can I help?",
    "tone": "casual"
  },
  "Humorous": {
    "greeting": "Sup, boss! Ready to take over the world?",
    "tone": "humorous"
  }
}
```

3. **Integration with OpenAI API**:
    - Pass the selected personality mode as part of the OpenAI API prompt (e.g., "Respond in a [formal/casual] tone").
    - Example prompt structure:

      ```
      text
      Copy code
      You are an AI assistant in "Humorous" mode. Greet the user informally and make light jokes when appropriate.
      ```

4. **Dynamic Personality Changes**:
    - Add commands for switching personality mid-conversation (e.g., "Be more serious" → switches to Professional mode).
    - Confirm the mode change with the user (e.g., "Understood! Switching to professional mode.").
5. **User Preferences**:
    - Allow Kyle to remember the preferred personality mode across sessions (e.g., "Default to 'Friendly' mode when I open the assistant.").

Implementation Details:

- Add a helper function:

  ```
  python
  Copy code
  def set_personality_mode(mode: str):
      # Load and apply personality settings based on mode
      ...
  ```

- Maintain a `current_personality` variable to store the selected mode.
- Integrate personality details into the OpenAI API payload (e.g., by adjusting the system prompt).

Output:

- A **step-by-step guide** for:
  1. Creating and loading the `personality.json` configuration file.
  2. Implementing personality switching within the assistant's main loop.
  3. Constructing OpenAI API prompts based on the selected personality mode.
- Provide **Python code snippets** for personality switching and updating:

```
python
Copy code
current_personality = "Friendly"

def apply_personality():
    with open("personality.json", "r") as f:
        personalities = json.load(f)
    return personalities.get(current_personality, {})

def set_personality_mode(mode):
    global current_personality
    current_personality = mode
    print(f"Personality mode set to {mode}.")
```

Optional Enhancements:

- Add **custom user-defined modes** (e.g., "Kyle, make your own mode where you're poetic").
- Add **emotional simulation** (e.g., happy, annoyed) that slightly changes response wording and speed.
- Implement a **fallback default mode** in case the user requests an unknown mode.

Visual Sharing (Screen Capture & Recognition)

Goal: I need my Python-based AI assistant, "Kyle," to capture and analyze what is displayed on my monitor in real time. It should be able to perform:

1. **Screen Capture**: Take full or partial screenshots on Windows at intervals (or on-demand).
2. **Text Recognition (OCR)**: Use a library (e.g., Tesseract, OpenCV) or cloud-based OCR service to extract text from these screenshots.
3. **Optional Object/Scene Detection**: If feasible, allow detection of specific UI elements or objects.
4. **Integration**: Provide a function that integrates with the main assistant code so it can:
    - Capture the screen.
    - Process/recognize text (or objects).
    - Return any relevant data to Kyle for context or next actions.

Requirements:

- Must run on Windows (using a virtual environment).
- Use Python libraries such as `pyautogui`, `mss`, or `PIL` for screen capture.
- Demonstrate how to install, set up, and call the OCR library (e.g., Tesseract or OpenCV).
- Provide code snippets with inline comments explaining:
    - How the capture intervals work (or on-demand capture).
    - How recognized text is returned to the main assistant process.
- Show me how to **limit** capture to a smaller area (for privacy/security).
- Include any **security considerations** (e.g., confirming user permission before capturing).

Output:

- A **step-by-step guide** for implementing screen capture and OCR in Python, plus sample code demonstrating how to integrate it into the assistant.

- Tips for improving accuracy and handling partial/rotated text.
- Best practices for real-time or near-real-time capture (frame rate, CPU usage, etc.).

Toggle Screenshare

Goal: Implement a **button** in the GUI that lets the user **enable or disable screen sharing** (screen capture/analysis). When “On,” Kyle can capture the current screen for OCR or other visual features; when “Off,” Kyle remains unaware of the screen contents.

Requirements:

1. **Screen Sharing Button**:

- Place a labeled button or toggle switch (e.g., “Screen Sharing: OFF/ON”) on the main GUI.
- When the user clicks to turn sharing **On**, Kyle’s screen capture function should start (e.g., start_screen_capture()).
- When turned **Off**, capture should immediately stop (stop_screen_capture()).

2. **Backend Integration**:

- **Create or integrate** functions in your code to capture the screen:
- start_screen_capture() begins a loop or timer-based captures (if using scheduled intervals).
- stop_screen_capture() halts captures and discards any queued images.
- Optionally, only capture on-demand if that’s the preferred approach.
- If you already have an OCR or image-processing pipeline, link it to these functions so it’s only active when Screen Sharing is ON.

3. **UI Feedback**:

- Display the **current status** (ON or OFF) in a label or change the button text accordingly.

- If screen sharing is **ON**, perhaps show a small indicator (icon or text) so the user knows Kyle can see the screen.

4. **Privacy & Security**:

- **Emphasize** that the user must explicitly enable screen capture.
- On first activation, show a short disclaimer or confirmation (e.g., “Kyle will now see your screen—proceed?”).
- Stop capturing completely when OFF, ensuring no background screenshots occur inadvertently.

5. **Implementation Details**:

- For a tkinter GUI, you can add something like:
- Bind this function to a button in your main window.
- If you rely on timed intervals, use after() or a scheduled job to manage periodic captures.

6. **Testing & Validation**:

- Verify that when the user toggles the button **ON**, screen capture or OCR starts working (e.g., Kyle can detect text on the screen).
- Confirm that turning it **OFF** fully halts captures—no further screenshots or OCR logs appear.
- Log or console messages can help confirm the state changes for debugging purposes.

Output:

- **Step-by-step code** or instructions demonstrating:

1. Adding the toggle button to the existing GUI layout.
2. Binding the button to start_screen_capture() and stop_screen_capture() functions.
3. Reflecting status changes in the interface (labels or button text).

- Provide an **example** or mockup screenshot showing how the button looks in the GUI.

Optional Enhancements:

- **Partial Capture**: Let the user define a specific region or window to capture instead of the entire screen.
- **Visual Indicator**: Blink or highlight the button while capturing to reassure users it's actively on.
- **OCR Integration**: If you have OCR or object detection in place, link that logic so the user can see Kyle's recognized text in real time.

Game analysing

Goal: Implement a new feature where Kyle can **watch a game**, capture relevant information (e.g., scoreboard, stats, player performance), and offer a **post-game breakdown**. Over time, Kyle should learn from each observation to **improve** its advice and analysis.

Requirements:

1. **Game Feed Capture**

- Allow Kyle to view the game screen in real time or via a recorded feed.
- Integrate screen capture or input-video handling:
- **Screen capture**: If the game is played locally, use the existing "screen sharing" feature or a specialized capture method for high-frame-rate contexts.
- **Video feed**: If you have pre-recorded sessions, read frames from a video file (e.g., OpenCV's VideoCapture).

2. **Real-Time / Post-Game Analysis**

- Detect and extract **key elements** (e.g., scoreboard info, player stats, item usage, map position) using techniques like:
- **OCR** for text-based stats (score, timer, etc.).
- **Object detection** (optional) for players, items, or map elements.
- Store these observations in an **internal dataset** or database for deeper analysis.

3. **Long-Term Learning**

- Integrate the extracted data into Kyle's **vector-based memory** or a specialized "game knowledge" database.

- Over time, the system should **recognize patterns** (e.g., strategies, best item usage) and refine its advice.
- Possibly incorporate a **machine learning** component that can process historical data to generate better insights (e.g., “When you pick Strategy X in these conditions, your win rate increases by 15%”).

4. **Post-Game Breakdown**

- Provide a **summary** or “highlight reel” after each session:
- “Score was 10-8, you performed best when staying near Objective B,” etc.
- Offer **advice** based on historical data:
- “You’ve won 80% of games when you pick Champion A, but only 40% with Champion B—consider practicing B more.”
- Output the breakdown in **plain conversational text** (no markdown formatting).

5. **User Control & Privacy**

- Add a GUI toggle or command for “Watch this game.”
- Once the user finishes or stops the session, Kyle should finalize and store the data.
- Provide an **option** for the user to delete or anonymize older sessions if desired.

6. **Implementation Details**:

- **Capture / Analysis**:
- Use OpenCV (or another library) to grab frames.
- Possibly run **OCR** (Tesseract, Google Vision, etc.) on scoreboard/text overlays.
- If advanced detection is needed, use a **model** (e.g., YOLO, Mask R-CNN) for object tracking in the game environment.
- **Data Storage**:
- Save crucial stats (e.g., final score, mid-game events, kills, items) in a structured format (SQLite or a vector database for semantic recall).

- Integrate these entries into Kyle's existing memory or a separate "game_data" memory module for specialized queries.
- **Learning / Pattern Recognition**:
- For more advanced learning, consider a **ML model** that ingests your aggregated game data to produce better insights or predictions.
- E.g., "If your approach is to push mid-lane early, your success rate is X% given certain champion matchups."

7. **Testing & Validation**:

- Test with a small set of recorded games to confirm that:
- The scoreboard/stats are recognized reliably.
- The summary includes correct context (score, major events, best plays).
- Over multiple sessions, confirm the system's ability to **reference** older data and produce improved advice.
- Evaluate performance overhead (capturing frames + analyzing them) and optimize if necessary (e.g., lower capture frequency or scale down resolution).

Output:

- **Step-by-step code snippets** or instructions showing how to:

1. Capture or load the game feed.
2. Detect and extract relevant stats (OCR, object detection).
3. Store the data in a structured format (integrating with the vector-based memory for further queries).
4. Generate a post-game breakdown, referencing both the current and historical data.

- Provide an **example** breakdown after a session, illustrating how Kyle uses historical context to refine its advice.

Optional Enhancements:

- Integrate **live commentary**: Kyle can provide real-time suggestions during the game, not just a post-game breakdown.

- Add a **coaching mode** that notifies the user of recurring mistakes or missed opportunities.
- Perform **multimodal analysis** combining game data with user's voice chat to detect synergy or issues in team communication.

Wake on voice

Goal: Allow Kyle to **listen for its name** (or a custom wake word) to start interacting, instead of requiring the user to press a button. Once the wake word is detected, Kyle should engage speech recognition and respond accordingly.

Requirements:

1. **Wake Word Detection**:

- Integrate a **hotword detection** library or service (e.g., **Snowboy**, **Picovoice Porcupine**, or a custom wake word model).
- The default wake word is Kyle's name (e.g., "Hey Kyle"), but allow for **configurable** options (e.g., "Hey Assistant!").
- Ensure minimal **false triggers** by tuning sensitivity thresholds or using multi-stage confirmation if needed.

2. **Continuous Listening & Activation**:

- Add an **idle listening** mode, where the system monitors the microphone for the wake word, using lightweight audio processing.
- Once the wake word is recognized, switch to **active** speech recognition mode:
- Example: "Hey Kyle!" → beep or short acknowledgement → user speaks a command → Kyle processes and responds.
- Return to **idle** mode after the user finishes speaking.

3. **Privacy & Performance Considerations**:

- Provide a way for the user to **turn off** continuous listening if desired (e.g., a GUI toggle or voice command "Stop listening for now.").
- Use an offline hotword detection engine (e.g., Snowboy or Porcupine) to avoid sending continuous audio to the cloud.

- Optimize CPU usage, especially if running on lower-end devices.

4. **GUI Integration** (Optional):

- Update the GUI to display an **indicator** (e.g., a colored dot or microphone icon) showing:
- Idle listening (wake word detection on).
- Active listening (currently capturing full speech for recognition).
- Provide a **switch** or checkbox for enabling/disabling the wake word feature.

5. **Implementation Steps**:

1. **Choose a Hotword Engine**:

- Examples:
- **Snowboy** (end-of-life as a service, but existing models can still be used).
- **Porcupine** (Picovoice), with a custom or built-in “Kyle” model.
- Alternatively, if using **Google Cloud** or another cloud-based engine, confirm it supports wake word detection.

2. **Load/Train a Wake Word Model**:

- If no pre-trained model exists, gather samples of “Hey Kyle” in different voices/noise conditions.
- Train or fine-tune a small model for better accuracy.

3. **Integrate with Speech Recognition**:

- Once the wake word is detected, temporarily switch from a “light listening” thread to the main speech recognition function (e.g., speech_recognition.Recognizer or a more robust engine).
- Provide a short beep/response (“Yes?” or “I’m listening...”) to let the user know it’s ready.

4. **Switch Back to Idle**:

- After the user’s command is processed, return to hotword-only detection to reduce overhead.

6. **Testing & Validation**:

- Test in quiet and noisy environments to ensure reliability.

- Adjust sensitivity or threshold to minimize misfires.
- Confirm that once the user says “Hey Kyle,” the system seamlessly transitions to listening for a full command.

Output:

- **Step-by-step code snippets** or instructions showing how to:

1. Install and configure the chosen hotword detection library.
2. Set up a continuous audio stream that monitors for the wake word.
3. Transition from idle wake word listening to active speech recognition.
4. Return to idle mode after the user’s command is processed.

- Provide **example** usage, e.g., “Hey Kyle! What’s the weather like?” → Kyle responds.

Optional Enhancements:

- Add a **custom phrase** (e.g., “Computer”) as an alternative wake word.
- Provide **visual feedback** (e.g., blinking icon) whenever Kyle hears a partial or near match to avoid confusion.
- Implement a simple **confirmation** if the confidence level is borderline (“I think you said ‘Hey Kyle.’ Should I listen now?”).

Additional features

Below are **twelve detailed prompts** you can provide to your developer agent for each feature idea. Each prompt outlines the **Goal**, **Requirements**, and **Implementation Details** or **Action Items**, ensuring clarity on how to integrate each feature into Kyle. Feel free to adjust them based on your specific workflow or existing codebase.

1. Routine / Daily Summary

Prompt: Routine / Daily Summary Feature

Goal: Have Kyle provide a **daily (or routine) summary** each morning (or at a user-defined time), including weather, calendar events, news snippets, and reminders.

Requirements:

1. **Scheduled Task**:

- Use a background scheduler (e.g., schedule library) or system cron job.
- Default time (like 7 AM) or user-configurable.

2. **Data Gathering**:

- Fetch data from integrated APIs (weather, news, reminders, calendar events).
- Summarize them into a concise text or speech output.

3. **Personalization**:

- User can specify which categories to include (weather, personal tasks, top headlines, etc.).

4. **GUI & Voice Output**:

- Show a "Daily Summary" popup in the GUI or speak it out loud if the user has voice mode active.

Output:

- Step-by-step code/examples demonstrating:

1. Setting up the scheduling mechanism.
2. Pulling relevant data (e.g., calling get_weather(), get_news_summary()).
3. Formatting the data into a coherent summary.

- Provide an example daily summary output in plain text ("Good morning! Today's weather is... You have 2 upcoming tasks...").

2. Calendar / Task Sync

Prompt: Calendar & Task Integration

Goal: Synchronize Kyle with popular calendars (Google Calendar, Outlook) and task managers so users can create or view events and tasks through voice or text commands.

Requirements:

1. **API Integration**:

- Use Google Calendar API or Microsoft Graph for Outlook.
- Store OAuth credentials securely.

2. **Voice Commands**:

- "Kyle, create an event next Tuesday at 2 PM: Team Meeting."
- "Kyle, what's on my schedule for tomorrow?"

3. **Task Management**:

- Extend to services like Todoist or Notion in the future.
- Commands like "Add grocery shopping to my tasks."

4. **Notifications**:

- Kyle can proactively remind you of upcoming events or tasks when it's time.

Output:

- Code snippets demonstrating authentication (OAuth flow), event creation, retrieval, and integration into Kyle's conversation loop.
- Example usage showing user voice commands → new calendar events appear.

3. Multi-User or Multi-Profile Support

Prompt: Multi-User Profile Implementation

Goal: Allow multiple users to use Kyle, each with their own preferences, reminders, and memory.

Requirements:

1. **User Identification**:

- Simple approach: user picks a profile on startup or states "I'm Alice" so Kyle switches context.
- More advanced: voice recognition or separate wake words.

2. **Data Separation**:

- Maintain separate SQLite databases, or partition memory entries by user_id.

3. **Personality & Settings**:

- Each user can have a default personality mode, accent settings, or default wake word.

4. **GUI**:

- A dropdown to select the current user, or a prompt upon launching Kyle.

Output:

- Step-by-step guide or code sample showing how to store user profiles, retrieve them, and handle user-specific data.
- Example usage showing “Switch to user John” → Kyle loads John’s memory/personality.

4. Custom Plugin Architecture

Prompt: Custom Plugin System

Goal: Create a **plugin architecture** so new functionalities (like custom commands or data sources) can be added without modifying Kyle’s core code.

Requirements:

1. **Plugin Interface**:

- Define a Python class or module structure (MyPlugin.py) with on_load(), on_command(command), etc.
- Kyle scans a plugins/ folder on startup, loads each plugin automatically.

2. **Core Registration**:

- Provide a way for each plugin to register commands or triggers with Kyle.

3. **Security / Isolation**:

- Possibly restrict plugin capabilities or require user approval.

Output:

- Sample plugin code plus a main system that auto-discovers plugins.
- Explanation of how to develop a new plugin (e.g., “WeatherPlugin” or “CryptoPricePlugin”).

5. IoT / Smart Home Integration

Prompt: IoT / Smart Home Control

Goal: Connect Kyle to smart home devices (lights, thermostat, etc.), enabling voice commands like “Turn off the kitchen lights.”

Requirements:

1. **Protocol / Platform Support**:

- Integrate with Home Assistant, Alexa Skills, or direct device APIs.
- Possibly set up a local Home Assistant server that Kyle queries.

2. **Voice Commands**:

- “Kyle, set the thermostat to 72 degrees.”
- “Lock the front door.”

3. **Feedback**:

- “Lights are now off.” or “The thermostat is set to 72.”

Output:

- Code showing how to authenticate with chosen IoT platform.
- Example usage with a simple device (smart bulb or thermostat) and how Kyle interprets the commands.

6. Interactive Learning / Tutoring Mode

Prompt: Interactive Tutoring Feature

Goal: Allow Kyle to act as a tutor, providing quizzes or lessons in certain subjects (languages, math, science).

Requirements:

1. **Lesson Content**:

- Could be from an external source (API) or a curated knowledge base.

2. **Adaptive Quizzing**:

- Keep track of the user’s answers, difficulty level, and adjust questions accordingly.

3. **Memory of Progress**:

- Store user’s performance in Kyle’s memory; recall it for personalized follow-up.

Output:

- Example code for a “quiz loop” (Kyle asks a question, user answers, Kyle provides feedback).

- Show how to store the user's performance data and adapt future questions.

7. Translation & Multilingual Support

Prompt: Multilingual / Translation Capabilities

Goal: Enable Kyle to translate text or speech into different languages, and respond in that language if requested.

Requirements:

1. **Translation API**:

- Use Google Translate API, DeepL, or a local model for offline.

2. **Voice Output**:

- Switch text-to-speech languages if needed.

3. **Language Code Detection**:

- Detect user's spoken language if the user switches to Spanish, for example.

Output:

- Code snippet for "Kyle, translate 'Hello, how are you?' into French."
- Explanation of how to change speech recognition language or TTS voices for multilingual replies.

8. Advanced Sentiment or Emotion Detection

Prompt: Emotion & Sentiment Analysis

Goal: Let Kyle detect user sentiment (happy, sad, stressed) in voice or text, and respond empathetically.

Requirements:

1. **Text Sentiment**:

- Use an NLP pipeline (Hugging Face sentiment analysis).

2. **Voice Emotion** (Optional advanced):

- Analyze acoustic features (pitch, energy) for emotional cues.

3. **Adaptive Responses**:

- If user is sad, Kyle uses a more comforting tone or suggests helpful actions.

Output:

- Steps for integrating a sentiment model and hooking it into Kyle's conversation logic.
- Examples of varied responses based on user emotional state.

9. Proactive Notifications & Alerts

Prompt: Proactive Alerts System

Goal: Kyle shouldn't only react; it should also alert the user about important events or changes (e.g., meeting in 10 minutes, or a stock price threshold reached).

Requirements:

1. **Event or Condition Monitoring**:
- Schedules, stock prices, system events, etc.
2. **Configurable Alerts**:
- User sets thresholds (e.g., "Alert me if my stock goes above $100").
3. **Notification Channels**:
- Display a popup in the GUI, speak the alert, or send an email/SMS.

Output:

- Code that demonstrates the scheduling or event-driven triggers.
- Example usage: "Notify me if the temperature in my house goes above 80°F."

10. Extended Memory Summarization

Prompt: Memory Summarization / Compression

Goal: Periodically summarize older conversation logs or data to reduce memory size while retaining key info.

Requirements:

1. **Summarization Model**:
- Could be a local model or an API call (OpenAI GPT).
2. **Periodic Summaries**:
- Summarize data older than X days.
- Replace detailed entries with a short summary in the vector-based memory.
3. **Retain Key Data**:

- Keep crucial facts, discard filler text.

Output:

- Example code for “summarize_old_memories()” that processes logs, updates the database with summarized text.
- Explanation of how to combine new and summarized data for conversation context.

11. Graphical or Visual Output

Prompt: Visual Data & Chart Generation

Goal: Let Kyle present charts, graphs, or images to the user in the GUI for data-based insights (e.g., game stats, habit tracking).

Requirements:

1. **Plotting Library**:
   - Use matplotlib, plotly, or seaborn to generate images.
2. **GUI Integration**:
   - Display generated plots in a popup window or embed them in the main interface.
3. **Voice Commands**:
   - “Kyle, show me my workout progress chart.”

Output:

- Step-by-step code with an example plot function (e.g., line chart of user’s daily steps).
- Explanation of how to embed the image in the GUI or provide a link if it’s a web-based interface.

12. Code Review & Debugging Assistance

Prompt: AI-Driven Coding Companion

Goal: Let Kyle assist with code reviews, debugging suggestions, or snippet explanations.

Requirements:

1. **Input Methods**:

- User can paste code, or point Kyle to a file.

2. **Parsing & Analysis**:

- Could rely on GPT code analysis or local linters/formatters for suggestions.

3. **GUI or Voice Commands**:

- “Kyle, review this function for efficiency.”
- Kyle provides plain-text feedback, suggestions, or possible refactors.

Output:

- Code samples for integrating code-linting or GPT-based analysis.
- Example scenario: user provides a snippet → Kyle identifies a bug or suggests improvements.

- ***How to Use These Prompts**

1. **Select a Feature**: Decide which one(s) align best with your immediate goals.

2. **Provide the Prompt**: Send it to your development agent or code-generation tool.

3. **Review & Customize**: Adapt the generated code or instructions to fit your existing architecture.

4. **Test Thoroughly**: Ensure each feature is stable before proceeding to the next.

With these **twelve detailed prompts**, you’re equipped to expand Kyle’s functionality across productivity, learning, IoT, advanced memory, and more, making the assistant a truly versatile companion.