Urdu LLM v02

1. Abstract

The Urdu Notebook Assistant v0.2 is an experimental system that combines **Streamlit** for user interaction with the **OpenAI API** for Urdu text summarization and question answering. This version builds on the earlier prototype by replacing placeholder functions with actual calls to a language model, giving the user real AI-powered responses. The app allows users to type chat messages, paste longer Urdu texts, request summaries, or ask questions about the text. All conversations are saved into a JSON file (`chats_v03.json`), which means the system remembers previous chats across sessions.

This prototype is intentionally simple, designed to explore how modern AI can be wrapped in a chat interface for Urdu language processing. It also demonstrates the typical workflow of building a research prototype: designing an interface, handling data persistence, connecting with an external API, and dealing with its limitations.

2. Program Overview

The program is structured around two pillars: **interaction** and **memory**. Interaction is managed through Streamlit, which provides the sidebar for chat history, the chat bubbles for user and assistant messages, and the input areas for text. Memory is managed through a JSON file that stores each chat session. Together, these allow a user to have multiple conversations, save them, return to them later, and continue where they left off.

What makes this version different from v0.1 is that it does not rely solely on simple rules for summarization. Instead, it sends the user's input to OpenAI's GPT model (`gpt-4o-mini`) and returns a generated response. This allows the assistant to provide more natural summaries and context-aware answers to Urdu questions.

The workflow goes like this: a user either writes a chat message or pastes a text. If it's a text, they can click on "Summarize" or "Ask Question." That input is passed to the OpenAI model through a carefully crafted **prompt**. The model responds, and that response is displayed back in the interface as if the assistant had spoken. Meanwhile, the program also saves both the user's message and the assistant's reply into the JSON file so that the conversation is not lost.

3. Data Types & Structures

Understanding the underlying structures is important because they explain how information is represented and stored.

Each chat session is assigned a unique ID using `uuid`. Inside that chat, there is a **title** (usually the first user message), a **timestamp** (when it was created or last updated), and a list of **messages**.

Each message itself is a dictionary with three key fields: the `role` (either `"user"` or `"assistant"`), the `content` (the actual Urdu text), and the `timestamp` in ISO format.

An example message looks like this:

```
{
    "role": "assistant",
    "content": "📄 **خلاصہ:** یہ متن کا خلاصہ ہے۔",
    "timestamp": "2025-08-29T04:01:19.288623"
}
```

These are all stored together in `chats_v03.json`. The file is essentially a dictionary of chats, where each chat is keyed by its unique ID. This format is simple and makes it easy to load and save entire histories in one step.

4. Functions

4.1 Persistence and State Functions

- `load_chats()` and `save_chats()` handle reading from and writing to the JSON file. If the file is missing or corrupted, `load_chats()` simply returns an empty dictionary so that the program does not crash.
- `save_current_chat()` updates the JSON file with the current session's messages. It automatically generates a title from the first user message so that chats are easier to recognize in the sidebar.
- `add_message(role, content)` appends a new message to the chat and ensures it is saved immediately.

4.2 Chat Management Functions

- `create_new_chat()` resets the session state with a new ID and clears previous messages.
- `switch_chat(chat_id)` loads messages from an existing chat, allowing the user to revisit old sessions.
- `delete_chat(chat_id)` removes a chat from memory and the JSON file. If the current chat is deleted, a new one is created automatically to avoid breaking the interface.

4.3 AI Functions

This is where v0.2 really changes from v0.1.

- `ai_summarize(text)` creates a **prompt in Urdu** asking the model to summarize the provided text. The prompt is crafted to encourage concise and accurate summaries. It then calls `client.chat.completions.create()` with `gpt-4o-mini`. The result is formatted with a "📄 خلاصہ" prefix.

- `ai_qa(text, question)` creates a structured Urdu prompt that includes both the text and the user's question. It then sends this to the model and returns the assistant's answer with a "🤖 جواب" prefix.

Both functions use a **temperature parameter** (0.3 for summarization, 0.2 for Q&A). This controls randomness. Lower values mean the model is more deterministic, which is important for consistent summaries and factual answers.

4.4 Support Functions

- `require_api_key()` checks whether the `OPENAI_API_KEY` environment variable is set. If not, it stops the app and displays an error message. This protects the program from running without proper credentials.

5. User Interface

The interface is built entirely in Streamlit. The page is configured with a title, icon, and a wide layout. CSS is injected to make the chat bubbles look like ChatGPT, with distinct styling for user and assistant messages.

The **sidebar** contains buttons for starting new chats, viewing saved chats, and deleting them. Each chat is displayed with its title and timestamp, and clicking on one switches the conversation.

The **main panel** has two modes:

1. **Chat Mode**: Users type a message and send it. The assistant responds with an acknowledgment (currently a placeholder in chat mode, though summarization and Q&A use GPT).
2. **Analysis Mode**: Users paste longer Urdu text and then choose to either summarize it or ask a question about it. If they ask a question, a secondary input box appears for typing the query.

This split design makes the app flexible: it can handle both short conversational turns and longer academic-style text analysis.

6. Limitations (My Perspective)

Even though this version uses OpenAI, I quickly realized that it is not without problems. For one, the model sometimes returns errors such as "429 – exceeded quota". This happens when usage limits are hit. It reminded me that APIs are powerful but not unlimited, and sometimes the assistant will fail because of external reasons outside my control.

Another improvement would be to expand the AI integration. While OpenAI is powerful, it comes with restrictions: it requires internet access, it has rate limits, and it depends on paid quotas. To make the assistant more reliable, I could explore **alternative APIs** or **open-source models** that

can be run locally on a server. For example, projects like **Hugging Face Transformers** or **Rasa** could provide Urdu summarization and Q&A without needing to connect to the internet. Running a local model on a GPU or a server would mean there are no quota errors and no risk of the service being unavailable. This could be especially useful in research or educational settings where internet access is limited or privacy is important.

I also want to improve user experience. For example, instead of showing static responses, the assistant could stream answers word by word, similar to ChatGPT. I could add file upload support (so users could analyze PDFs, Word documents, or plain text files) and also provide export options (such as saving chats as PDFs for academic use). These features would make the assistant more practical and closer to a real academic or research tool.

Most importantly, I want to keep refining how the assistant understands prompts. Prompt engineering plays a huge role in the quality of summaries and answers. By experimenting with different instruction styles and models, I can make the assistant more accurate and trustworthy for Urdu text analysis.

7. Learning Outcomes (My Perspective)

When I first started working on this project, it honestly felt daunting. I had never built something that combined so many moving parts: a web app, saving chat histories, calling an external AI API, and dealing with state across different sessions. At the beginning, I wasn't sure if I could actually connect everything without breaking it. But step by step, the process became less intimidating as AI helped me along the way, I found useful courses online from websites like Coursera and Stackflow Exchange. Whenever I got stuck, I experimented, asked questions, and realized that even the assistant I was building could become part of my learning process.

One of the biggest areas of growth for me was understanding Streamlit. Before this, I only knew Python as a scripting language that runs in a terminal. Suddenly, I was building something that looked and felt like a modern application. Learning about session states, chat layouts, and CSS styling for chat bubbles taught me how much presentation matters when building tools. This was new territory, but it gave me confidence that I could actually build interfaces people might want to use.

I also learned a lot about data persistence and storage. At first, saving chats to a JSON file felt simple enough, but I quickly realized how powerful even that small step was. It made me appreciate the idea of "memory" in applications, that data doesn't disappear when you close the app. At the same time, it opened my eyes to the limits of JSON and why databases are used in real systems. I didn't implement a database yet, but I now understand why it matters.

Another area I learned about was natural language processing and AI integration. My first attempt at summarization and Q&A was completely rule-based and very limited. Switching to the OpenAI API in v0.2 was my first real experience calling an external model. This taught me about prompts, temperature settings, and how different wording can completely change the outcome of a model's answer. I also learned that even big AI systems are not perfect — sometimes they fail with errors (like quota exceeded), and that forced me to think about error handling and reliability.

This project also gave me a taste of software engineering practices. Using a .env file to store my API key showed me how developers protect sensitive information. Designing functions that are reusable (like add_message() or ai_summarize()) taught me how to build modular code.

Finally, I think the most valuable thing I learned is about my own learning process. At first, I wanted to jump straight into making something "smart." But I realized that even a simple prototype has value if I can explain it clearly, identify its limitations, and show how it could grow. That mindset shift from "I have to build something perfect" to "I have to build something that demonstrates ideas" is probably the most important outcome for me as someone aspiring to be a research assistant.

8. Future Improvements

In the future, I would like to make the chat system more robust. For example, instead of just saving chats locally, I could move to a database so that multiple users can have private sessions. I also want to add error handling that gracefully informs the user when the API is unavailable, perhaps with offline fallback summaries.

I could also expand the assistant's abilities: allowing file uploads (so users can summarize PDFs or Word documents), adding export options (so users can save chats as PDF or Word), and introducing streaming responses so that the assistant's reply appears word by word. These improvements would make the prototype feel even closer to real AI applications while keeping Urdu support at the center.

9. Improvements from the First Iteration

Looking back at my first prototype (v0.1), it was essentially a simple form where I pasted Urdu text, and the assistant gave back a fixed response or a basic two-sentence summary. That was useful for getting the workflow right, but it wasn't very intelligent. In **this version (v0.2)**, I improved things by making the interface more natural and chat-like, so it feels closer to modern AI assistants. I also added the ability to save multiple chats and revisit them, which makes the tool feel like a real notebook rather than a one-off script.

The biggest step forward is that I integrated the **OpenAI API**. This meant moving away from my placeholder `simple_summarize` and `simple_qa` functions, and actually connecting the app to a powerful language model. I learned how to store my API key securely in a `.env` file (instead of hardcoding it), which is good practice for real applications. Using `.env` files also made me realize how researchers and developers usually protect sensitive information.

Even though I connected OpenAI, I noticed something important: the service itself has **limitations**. For example, in my saved chats, you can see that the assistant sometimes returned errors like "429 – exceeded quota." This shows that even professional APIs have constraints, like rate limits, billing issues, or downtime. I think this is a valuable lesson: adding a big AI model doesn't magically solve everything. It taught me to expect and design around errors.

In the future, I can imagine improving this further by not just calling the API, but also adding **fall-back options** (like a smaller offline model) so that the assistant doesn't completely fail when the API is unavailable. This is something I would not have understood in v0.1, but now I see how important robustness is.

10. Disclaimer and Future Portability

I also want to make it clear that I chose **Streamlit** because it is excellent for quickly building prototypes. It allowed me to go from an idea to a working Urdu chatbot in very little time. However, I understand that Streamlit is not always the best choice for long-term deployment. It usually requires the terminal to stay open, and it is more research-friendly than production-ready. If this project grows, I may need to "graduate" to something more portable, such as a Flask or FastAPI backend combined with a React front end, or even a mobile app. For now, though, Streamlit lets me learn and iterate fast, which is exactly what I need at this stage.