**Overall Workflow:**

The application follows this sequence:

1. **Setup & Initialization:** Load configurations, prompts, and initialize the Streamlit UI and session state.
2. **User Input:** Allow the user to upload documents via the sidebar and edit the initial prompt.
3. **Initial Generation (Triggered by Button):**  
   a. Format uploaded documents and the first prompt.  
   b. **LLM Call 1 (Generator):** Send the formatted prompt to the local Ollama model to generate a *draft* of testable items, strategy, scenarios, and cases.  
   c. Parse the output of Call 1 to separate the extracted items from the draft artifacts.  
   d. Format the second prompt using the parsed output from Call 1 and the original documents.  
   e. **LLM Call 2 (Refiner):** Send the second prompt to Ollama to refine the artifacts, ensure coverage using original docs, and generate the final Testable Items list, Strategy, Scenarios, Cases, and Traceability Matrix.  
   f. Store the final, refined output from Call 2.
4. **Display Initial Results:** Show the refined artifacts generated by Call 2 in a dedicated section.
5. **Conversational Follow-up:**  
   a. Enable a chat interface.  
   b. User asks follow-up questions or requests modifications.  
   c. Format the *entire* conversation history (including the initial refined output and subsequent turns) into a single prompt string.  
   d. **LLM Call 3+ (Chat):** Send the formatted history prompt to Ollama.  
   e. Display the LLM's response and update the history.
6. **New Session:** Allow the user to reset everything and start over.

**Code Breakdown Step-by-Step:**

1. **Imports and Logging Setup (Lines 1-19):**
   * import streamlit as st, json, os, uuid, datetime, dotenv, requests, logging: Imports all necessary libraries.
   * logging.basicConfig(...): Configures the logging system to print INFO level messages (and above) to the console with a specific format.
   * logger = logging.getLogger(\_\_name\_\_): Creates a logger instance for use throughout the script.
2. **Configuration & Initialization (Lines 21-30):**
   * load\_dotenv(): Loads variables from your .env file (like OLLAMA\_BASE\_URL, OLLAMA\_MODEL\_NAME).
   * logger.info("Environment variables loaded."): Logs that the loading attempt was made.
   * OLLAMA\_BASE\_URL = ..., OLLAMA\_API\_ENDPOINT = ..., OLLAMA\_MODEL\_NAME = ...: Sets up the connection details for your local Ollama instance, using defaults if environment variables aren't found.
   * logger.info(f"Ollama configured..."): Logs the configuration being used.
3. **Load Prompts from Files (Lines 32-55):**
   * load\_prompt\_from\_file(filepath, ...) function:
     + Takes a file path as input.
     + Logs the attempt to load the file.
     + Uses a try...except block to handle FileNotFoundError and other potential reading errors.
     + Logs success or error messages.
     + Returns the file content as a string or a default value on error.
   * PROMPT\_TEMPLATE\_CALL\_1\_BASE = ..., PROMPT\_TEMPLATE\_CALL\_2\_BASE = ...: Calls the function to load the content of prompt\_call\_1.txt and prompt\_call\_2.txt into variables.
   * if not PROMPT\_TEMPLATE\_CALL\_1\_BASE: ...: Checks if the prompts loaded successfully. If not, logs a critical error and stops the Streamlit app using st.stop() because these prompts are essential.
4. **Helper Function: call\_llm\_service (Lines 57-97):**
   * This is the core function for interacting with Ollama.
   * def call\_llm\_service(prompt: str) -> str | None:: Defines the function taking a single prompt string and returning either a string response or None on error.
   * logger.info(...): Logs that a prompt is being sent and its length.
   * headers = ..., payload = ...: Creates the necessary HTTP headers and the JSON payload required by Ollama's /api/generate endpoint, including the model name and the input prompt. stream is False to get the full response at once.
   * try...except: Wraps the API call in error handling.
   * response = requests.post(...): Makes the actual HTTP POST request to the Ollama API endpoint with the payload and a timeout (set to 3000 seconds).
   * response.raise\_for\_status(): Checks if the HTTP response indicates an error (like 404 Not Found if the model isn't available, or 500 Internal Server Error).
   * response\_data = response.json(): Parses the JSON response from Ollama.
   * llm\_response = response\_data.get("response", ""): Extracts the actual generated text from the JSON response (Ollama uses the "response" key).
   * logger.info(...): Logs that a response was received and its length.
   * return llm\_response.strip(): Returns the cleaned-up response text.
   * except requests.exceptions...: Catches specific errors like connection issues, timeouts, general request problems, and JSON parsing errors, logs them, displays an error in the Streamlit UI using st.error(), and returns None.
5. **Helper Function: parse\_generator\_output (Lines 99-126):**
   * Takes the raw string output from the first LLM call (generator\_output).
   * Logs that parsing is starting.
   * Initializes empty strings for items\_text and artifacts\_text.
   * Uses output\_text.split('## ') to break the text into parts based on the markdown headings.
   * Iterates through the parts, checking (case-insensitively) if a part starts with the expected headings ("extracted testable items", "draft test strategy", etc.).
   * Reconstructs the text for the items\_section and combines the strategy, scenarios, and cases into artifacts\_text.
   * Logs a warning if parsing seems incomplete.
   * Returns the two separated strings (items\_text, artifacts\_text). Includes error handling and fallback logic.
6. **Helper Function: format\_docs\_for\_prompt (Lines 128-139):**
   * Takes the list of uploaded file dictionaries (st.session\_state.uploaded\_files).
   * Logs how many documents are being formatted.
   * Iterates through the list.
   * For each document, creates a formatted string including the user-provided tag and the file content.
   * Joins these formatted strings together with separators (---).
   * Logs the total length of the formatted string.
   * Returns the single string containing all tagged document content.
7. **Helper Function: format\_history\_for\_local\_llm (Lines 141-150):**
   * Takes the chat history list (st.session\_state.messages).
   * Logs how many messages are being formatted.
   * Iterates through the messages.
   * Builds a single string, prefixing each message's content with "User:" or "Assistant:".
   * Appends "Assistant:\n" at the very end to prompt the LLM for its next response in the conversation.
   * Logs the length of the formatted history string.
   * Returns the single prompt string representing the conversation.
8. **Streamlit App UI Setup (Lines 152-171):**
   * st.set\_page\_config(...): Sets the page layout to wide.
   * st.markdown("<h1 style='text-align: center;'>...</h1>", ...): Displays the main title, centered using HTML within Markdown.
   * if 'session\_id' not in st.session\_state:: Initializes session state variables *only if they don't already exist*. This preserves state across reruns. Variables include:
     + session\_id: A unique ID for the current session.
     + messages: A list to store the chat history (dictionaries with "role" and "content").
     + uploaded\_files: A list to store uploaded document info (dictionaries with "tag" and "content").
     + prompt1\_text: Stores the text for the first prompt (initialized from the loaded file, editable by the user).
     + generation\_done: A boolean flag set to True after the initial 2-call generation succeeds.
     + refined\_output: Stores the raw output of Call 2 (though it's primarily accessed via st.session\_state.messages[0] now).
   * Logs the start of a new session.
9. **Sidebar UI (Lines 173-201):**
   * with st.sidebar:: Defines the content of the sidebar.
   * st.header, st.caption: Display titles.
   * doc\_tag = st.text\_input(...): Text input for the document tag.
   * uploaded\_file = st.file\_uploader(...): File uploader widget allowing .txt and .md files.
   * if st.button("Add Document", ...): Logic executed when the button is clicked.
     + Checks if both a tag and file are provided.
     + Reads the file content using uploaded\_file.getvalue().decode(...), including fallback decoding.
     + Appends the tag and content to st.session\_state.uploaded\_files.
     + Logs the addition and shows a success message.
     + Handles errors during file reading.
   * st.subheader("Uploaded Documents:"): Displays the list of tags for uploaded documents.
   * if st.button("✨ New Chat Session", ...): Logic for the reset button.
     + Logs the action.
     + Resets all relevant session state variables back to their initial states.
     + Generates a new session\_id.
     + st.rerun(): Forces Streamlit to rerun the script from the top, reflecting the cleared state.
10. **Main Area - Section 1: Prompt Editor (Lines 203-210):**
    * st.subheader, st.caption: Display titles.
    * prompt1\_edited = st.text\_area(...): Creates an editable text area, pre-filled with the content of st.session\_state.prompt1\_text.
    * st.session\_state.prompt1\_text = prompt1\_edited: Updates the session state with any edits made by the user in the text area.
11. **Main Area - "Generate Initial Artifacts" Button Logic (Lines 212-255):**
    * if st.button("Generate Initial Artifacts", ...): This block executes only when the button is clicked.
    * Logs the button click.
    * if not st.session\_state.uploaded\_files:: Checks if documents have been uploaded; shows a warning if not.
    * Resets state variables specific to a generation run (generation\_done, messages, refined\_output).
    * with st.spinner(...): Shows a loading spinner while processing.
    * tagged\_docs\_section = format\_docs\_for\_prompt(...): Calls the helper function to get the formatted document string.
    * final\_prompt\_call\_1\_content = ...: Formats the (potentially edited) prompt template from st.session\_state.prompt1\_text by inserting the tagged\_docs\_section.
    * logger.info("Initiating Local LLM Call 1..."): Logs the start of the first call.
    * generator\_output = call\_llm\_service(...): **Executes LLM Call 1.**
    * if generator\_output:: Checks if Call 1 was successful.
      + Logs completion.
      + items\_text, artifacts\_text = parse\_generator\_output(...): **Parses the output of Call 1.**
      + Handles potential parsing issues with logging and fallback.
      + logger.info("Initiating Local LLM Call 2..."): Logs the start of the second call.
      + final\_prompt\_call\_2\_content = ...: Formats the second prompt template using the parsed items\_text, artifacts\_text, and the original tagged\_docs\_section.
      + refined\_output\_text = call\_llm\_service(...): **Executes LLM Call 2.**
      + if refined\_output\_text:: Checks if Call 2 was successful.
        - Logs completion.
        - Stores the result in st.session\_state.refined\_output (though less critical now).
        - **Crucially, adds the entire refined output as the first message in the chat history:** st.session\_state.messages = [{"role": "assistant", "content": refined\_output\_text}].
        - Sets st.session\_state.generation\_done = True.
        - Shows a success message (st.success(...)).
        - Logs success.
        - st.rerun(): Reruns the app to display the results and enable the chat.
      + Handles Call 2 failure with logs and st.error.
    * Handles Call 1 failure with logs and st.error.
12. **Main Area - Section 2: Generated Artifacts Display (Lines 257-275):**
    * st.subheader("Generated Artifacts"): Displays the section title.
    * if st.session\_state.generation\_done or st.session\_state.messages:: Checks if generation is complete or if chat has started.
      + artifact\_container = st.container(...): **Creates the container only if artifacts should be shown.**
      + with artifact\_container:: Defines the content within the container.
      + if st.session\_state.messages and st.session\_state.messages[0]["role"] == "assistant":: Checks if the first message exists and is from the assistant (this holds the refined output).
        - st.markdown(...): Displays the refined output inside the container.
      + Handles the brief intermediate state after generation.
    * else:: If generation hasn't happened yet.
      + st.caption(...): Displays the placeholder text directly under the subheader (no container).
13. **Main Area - Section 3: ChatBox (Lines 277-304):**
    * st.subheader("ChatBox"): Displays the section title.
    * if st.session\_state.generation\_done:: Only enables the chat functionality *after* initial generation is complete.
      + if len(st.session\_state.messages) > 1:: Checks if there are any follow-up messages (more than just the initial assistant output).
        - chat\_history\_container = st.container(...): Creates a scrollable container for the chat history.
        - Iterates through st.session\_state.messages, *skipping the first message* (if i == 0: continue).
        - with st.chat\_message(...): Displays each user/assistant message using Streamlit's chat elements.
      + user\_chat\_prompt = st.chat\_input(...): Creates the chat input box at the bottom.
      + if user\_chat\_prompt:: Executes when the user types something and hits Enter.
        - Logs the user's question.
        - st.session\_state.messages.append(...): Adds the user's message to the history list.
        - history\_prompt\_string = format\_history\_for\_local\_llm(...): **Formats the entire history into a single string.**
        - with st.spinner(...): Shows a spinner during the LLM call.
        - logger.info("Sending follow-up..."): Logs the call.
        - response = call\_llm\_service(...): **Executes LLM Call 3+ (Follow-up).**
        - if response:: If successful.
          * Logs the received response.
          * st.session\_state.messages.append(...): Adds the assistant's response to the history.
        - Handles LLM call failure during chat.
        - st.rerun(): Reruns the app to display the newly added user message and assistant response in the chat history container.
    * else:: If generation isn't done yet.
      + st.caption(...): Shows a message indicating chat is disabled.