Web Research Agent: Expanded Documentation
This document provides a comprehensive overview of the Web Research Agent's operation, covering its end-to-end workflow, decision-making processes, error handling mechanisms, and system architecture.

1. End-to-End Workflow: A Detailed Plan
The Web Research Agent processes user queries through the following sequence of steps:

1. **Receive Query:** The user enters their research query via the Streamlit interface (located in `app.py`). This acts as the primary entry point for user interaction.
2. **Initiate Research:** The `app.py` script then triggers the research process by calling the `research()` method of the `WebResearchAgent` instance, which resides in the `research_agent.py` file.
3. **Analyze Query (`_analyze_query`):**
    - The agent begins by constructing a detailed prompt. This prompt is specifically designed for the Gemini LLM (Large Language Model). It instructs the LLM to thoroughly analyze the user's query, determining the intended purpose, the type of inquiry, the key topics involved, and to suggest optimal search terms and strategies.
    - This prompt is then sent to the Gemini LLM using the `call_real_llm` function, which handles the API interaction.
    - The LLM's response is carefully parsed. The agent extracts key pieces of information such as the intent, type of query, specific topics, proposed search terms, whether a news-focused search is required, and the recommended depth of the scraping process. Robust error handling is implemented at this stage to deal with potential parsing issues, ensuring that if any element cannot be extracted, default values are used to keep the process moving.
4. **Execute Research Plan (`_execute_research_plan`):**
    - Based on the LLM's analysis, the agent determines how many pages of search results to explore.
    - Using the `real_web_search` function (which leverages DuckDuckGo search), the agent performs web searches with the suggested terms.
    - If the analysis indicates a need for up-to-date information, the `search_for_news` function is used to conduct a news-specific search.
    - Search results from both web and news searches are gathered and consolidated. Duplicate results are eliminated to avoid redundancy.
    - A basic relevance check is performed. The agent scans the titles and snippets of the search results to ensure they contain the keywords identified by the LLM. While simple, this step helps filter out clearly irrelevant links. (Note: A more advanced relevance assessment using the LLM could be implemented in the future.)
    - For each relevant, unique link:
        - A short delay is introduced (`time.sleep`) to be respectful to web servers.
        - The content is then scraped using the `real_web_scraper` function, which utilizes the `requests` library and `BeautifulSoup` for HTML parsing. Error handling is critical here to manage timeouts, non-HTML content, or any

parsing errors that may occur.
        - Successfully scraped content is stored for later use.
    - News search results are kept in the form of snippets, rather than trying to retrieve the full content of each news article.
5. **Synthesize Information (`_synthesize_information`):**
    - The agent checks if any information has been successfully collected, either through scraping or news search. If no content is available, a message is prepared to indicate the lack of data.
    - A detailed prompt is constructed for the Gemini LLM. This prompt includes the original user query and all collected information (both scraped text and news snippets). The amount of scraped text provided to the LLM may be limited to prevent exceeding the LLM's context window.
    - The LLM is instructed to create a comprehensive report that directly answers the user's query. It is explicitly told to use only the information it is provided and to synthesize it into a clear, well-structured response.
    - The `call_real_llm` function is then used to send this prompt to the Gemini LLM. Again, robust error handling is in place to catch any issues with the API call or potential safety blocks from the LLM.
6. **Return Report:** The `research()` method returns either the generated report or an error/no-content message, depending on the outcome of the research and synthesis steps.
7. **Display Result:** Finally, `app.py` receives the report and presents it to the user through the Streamlit interface.

2. Decision-Making Processes
Throughout the workflow, the agent makes several key decisions:

1. **Query Understanding:** The first and perhaps most critical decision is performed by the LLM during the `_analyze_query` stage. Based on the input query, the LLM must decide:
    - What is the core intent of the user's query?
    - What type of question is it? (Factual, news-related, comparative, etc.)
    - What are the key topics involved?
    - What are the best search terms to yield relevant results?
    - Is a separate news search needed for up-to-date information?
    - How many search results (scrape depth) should be investigated?
2. **Tool Selection:** Based on the LLM's analysis, the agent selects which tools to use. Generally:
    - `real_web_search` is always used to perform a broad web search.
    - `search_for_news` is employed only if the LLM indicates a necessity for recent news or updates.
    - `real_web_scraper` is used on all potentially relevant links identified in the search results.
3. **Relevance Filtering:** Before scraping, the agent does a simple relevance check. It looks for the keywords identified by the LLM in the titles or snippets of search results. Links that do not seem to match these keywords are skipped to save processing time.

4. **Scraping Success:** The `real_web_scraper` function determines if scraping is technically possible (is the content HTML? Did the website respond?). It also assesses if any meaningful text was extracted. If scraping fails, the function returns `None`, and the agent moves on to the next link.
5. **Synthesis Context:** The agent verifies if any usable content (scraped text or news snippets) was gathered before attempting to synthesize a report. If no information is available, the synthesis step is skipped, and a message informing the user about the lack of data is returned.
6. **Final Report Generation:** The LLM makes the final major decisions during the `_synthesize_information` step. It determines the structure of the report, which information to include, and how to phrase the answer based on the user's query and the collected context.

3. Handling Problems & Edge Cases
The agent is built to handle various potential issues:

- **Unreachable Websites/Timeouts:** The `real_web_scraper` employs a timeout in its `requests.get()` call and uses `try...except requests.exceptions.RequestException`. If a website is unreachable or takes too long to respond, this exception is caught, an error message is printed to the console, and the scraping for that URL is skipped.
- **Scraping Failures:** Several types of scraping failures are handled:
    - The scraper checks the `Content-Type` header and ignores non-HTML content.
    - HTTP errors such as websites blocking scraping attempts (e.g., returning 403 Forbidden) are handled using `response.raise_for_status()`.
    - `BeautifulSoup` parsing errors are caught with a general `except Exception`.
    - In all these cases, an error is logged, and the agent moves to the next link.
- **Conflicting Information:** The agent relies on the LLM to manage conflicting information. The synthesis prompt requests that the LLM note any contradictions it finds in the provided data.
- **LLM API Errors:** The `call_real_llm` function uses `try...except Exception` around the `genai` API call. If the API fails (e.g., due to network issues, invalid key, or exceeding quotas), an error message is returned, and the overall operation fails gracefully.
- **LLM Safety Blocks:** The agent checks `response.prompt_feedback.block_reason` after each LLM call. If the prompt or response is blocked by safety filters, a descriptive error message is returned to the user instead of the intended result.
- **Insufficient Information:** As mentioned, if no usable data is collected, the synthesis step is skipped, and a message is displayed to inform the user that not enough information was found.
- **API Key Issues:** The code checks for the `GEMINI_API_KEY` at the start and within `call_real_llm`. If the key is missing, errors are printed, and LLM calls fail immediately.

4. Agent Structure & Architecture

The agent consists of:

- **`app.py` (Frontend):** Built with Streamlit. Provides