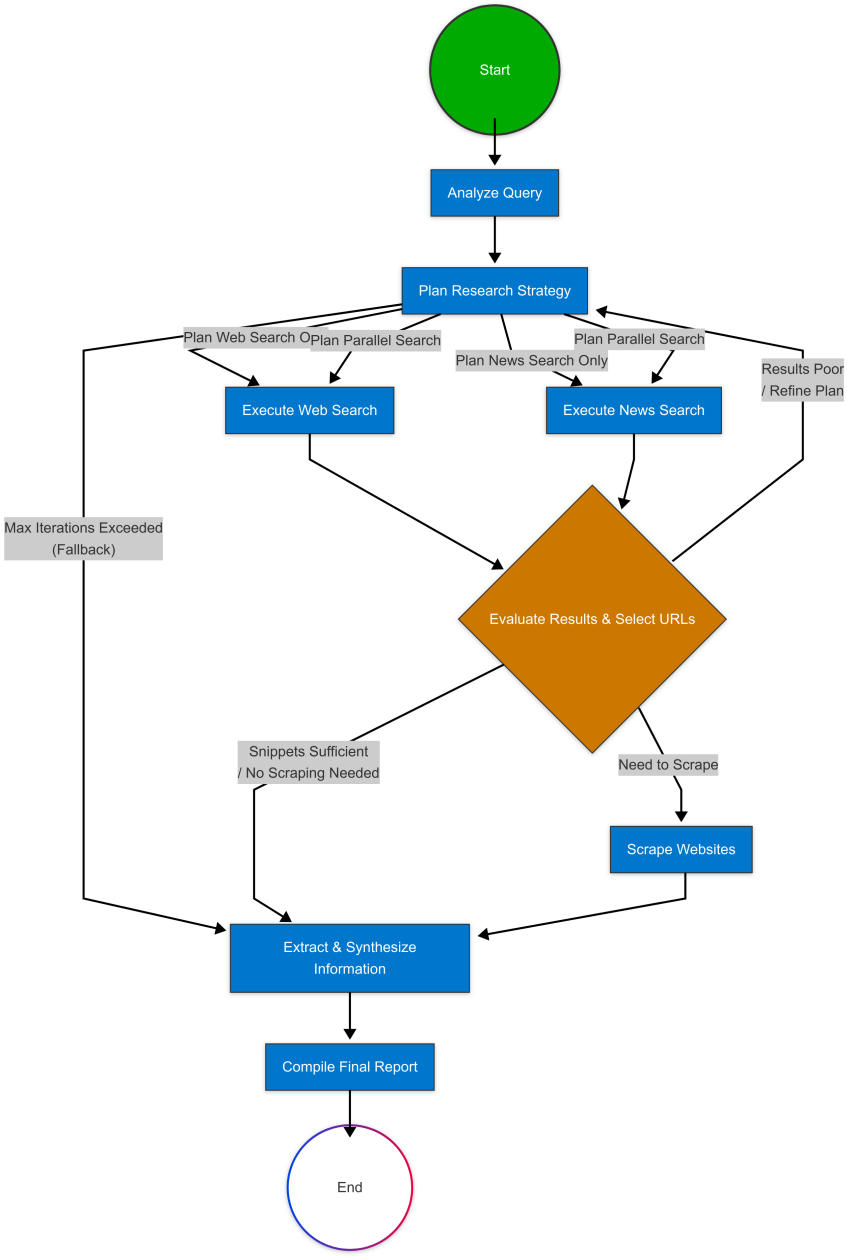
Agent Design & Architecture

My agent operates by guiding a query and accumulating research findings through a carefully designed flow managed by LangGraph. When a user submits a research query, the process immediately begins by initializing a state object that holds the original question. This state is then passed to the first node in our graph, the analyze_query node. Here, the agent employs gemini model to deeply understand the user's intent, dissect complex questions into core components, identify relevant entities, and determine the fundamental type of information required, be it recent news, historical data, or specific facts. The insights gained from this initial analysis, such as identified topics and required information types, are then added to the shared state, enriching our understanding of the task at hand.



With the query well understood, the state moves to the plan_research_strategy node. Leveraging the analytical results now present in the state, the LLM is prompted to formulate a strategic research plan. This plan is crucial; it dictates the most effective approach for gathering information, specifying which search tools (general search or news) are necessary, if searches should run in parallel, potential search terms, and outlining subsequent steps like scraping or search refinement. This strategic roadmap is recorded within the state, guiding the agent's next actions.

Following the plan, the agent enters the information discovery phase. Depending on the strategy formulated, the state is routed to either the execute_web_search, execute_news_search, or both simultaneously. These nodes interact with specialized external search tools (TavilySearchTool, NewsAggregatorTool) using the search terms derived from the state. The tools return relevant links, titles, and brief snippets, which are then seamlessly added back into the accumulating state.

The agent then proceeds to the evaluate_results_and_select_urls node. Here, the gathered search results are critically assessed. The agent, potentially with LLM assistance, evaluates the snippets to gauge if enough information has been found to answer the query directly. If not, it intelligently selects the most promising URLs from the search results for a more in-depth look. This evaluation node is also a key decision point; it determines the very next step in the workflow: either

proceeding to scrape the selected sites, jumping directly to synthesis if the snippets were sufficient, or routing back to the planning stage if the initial search results were poor and a refinement is needed. This decision is recorded in the state, directing the subsequent graph transition.

If the evaluation determined that more detail was necessary, the state moves to the scrape_websites node. This node utilizes an asynchronous web crawler (crawl4ai), which is enhanced with an LLM-driven strategy to accurately extract the main content from the selected URLs. This deep content is added to the state, providing the agent with the full text from relevant sources.

Regardless of whether scraping occurred, the state eventually reaches the extract_and_synthesize_information node. This is where the core intelligence of the agent shines in processing information. The LLM is provided with all the raw data gathered so far – search snippets, news articles, and scraped content. It meticulously synthesizes this diverse information, combining facts, identifying patterns, and structuring the findings into a coherent JSON output. This structured synthesis includes key findings, confidence levels, supporting evidence, and flags any potential gaps or contradictions encountered.

Finally, with the structured findings in hand, the state transitions to the compile_final_report node. This final node prompts the LLM to take the structured JSON synthesis and transform it into a polished, well-organized Markdown report. This involves formatting the content into a professional document complete with a clear title, an introduction, distinct sections detailing the findings, and potentially sections explaining the methodology or limitations of the research.

Addressing Errors:

Under the hood, the research agent is designed to gracefully handle situations when something goes wrong. If a search API is temporarily down, a specific web page refuses to load, or even if different sources present conflicting information, the agent is equipped to manage these issues. It achieves this by catching errors at various stages in the process, logging a brief note associated with the problematic URL or API call for transparency, and continuing its workflow using all the information it *can* successfully gather.

After collecting search snippets, it asks itself whether those summaries are enough. If there's nothing to work with or if the question calls for judgment or comparison, it will automatically circle back and try a refined search.

When it does reach out to scrape full pages, each URL is handled independently: a timeout or HTML parsing error on one site simply gets logged and skipped, rather than bringing the whole process down.

In its final report it will clearly call out any "limitations" or "low-confidence" flags so you know exactly where and why it had to guess.

In practice that means unreachable servers, missing data, or conflicting viewpoints won't crash the run. Instead, they reduce the agent's confidence score, trigger a smart retry or fallback, and get recorded in the final output. You always end up with a concise report.

Agent Structure

```
├── agent.py
├── api.py
├── main.py
├── nodes.py
├── tools.py
├── config.py
├── state.py
├── visualize_mermaid.py
```

agent.py: Configures and wires together the LangGraph StateGraph nodes and routing logic.
api.py: Exposes the research agent as a API service with /research endpoint.
main.py: Provides a command-line interface (and interactive mode) for running the agent and saving reports.
nodes.py: Implements each research step—query analysis, planning, web/news search, scraping, synthesis, and report generation.
tools.py: Wraps external web and news APIs (Tavily, NewsAPI) with built-in error handling and mock fallbacks.
config.py: Central configuration for API keys, model parameters, timeouts, and iteration limits.
state.py: Defines the ResearchState TypedDict schema shared across all nodes.
visualize_mermaid.py: Utility to generate a Mermaid diagram of the agent's workflow graph.