Technical Write-Up

1. Agent Style Choice

ReAct (ZERO_SHOT_REACT_DESCRIPTION)

For the creation of the Autonomous Fact-Checking Agent, the ReAct (Reasoning + Acting) agent architecture has been used instead of a typical ToolingAgent solution. The reason behind this selection is the task nature itself, which requires multi-step reasoning, dynamic planning, and adaptive action performance.

In a fact-checking process, the agent not only needs to obtain information but also iteratively generate verification strategies, assess contradictory evidence, and create subtle conclusions. ReAct agents naturally facilitate such a process through the integration of structured reasoning steps with tool application in an interleaved fashion so that the agent can reason its way through verification paths, alter its strategy on the basis of intermediate results, and keep track of contextual states over a sequence of actions.

In contrast, a ToolingAgent would execute stand-alone, stateless tool calls without having the facility to interact in sophisticated decision-making or adaptive planning, and this would heavily restrict the depth and accuracy of fact-checking outcomes.

The OpenAI LLM API (gpt-3.5-turbo / gpt-4)I performed multi-hop reasoning, evidence integration, and dynamic step handling with high fidelity, low latency, and good contextual retention even in deep multi-article fact-checking pipelines. Prompt engineering was optimized to trade off cost (token consumption) and accuracy, and the LLM was always able to maintain coherent flow across reasoning steps — an ability that was significantly enhanced by the transparent "Thought → Action → Observation" loops of the ReAct architecture.

2. Graph Schema Design

We implemented a lightweight in-memory graph (GraphManager) with three node types and three edge types:

Node Type	Properties
Claim	text (the original claim string)
Document	url (source URL), text (trimmed article content)
Snippet	summary, label (supports/refutes), score

Edge Relation	Meaning
cites	Claim → Document (this source was retrieved for the claim)
supports	Claim → Snippet (this snippet supports the claim)
refutes	Claim → Snippet (this snippet refutes the claim)

- **Why in-memory?** For rapid iteration and a stateless CLI/Streamlit demo, we avoided external dependencies (SQL, Neo4j, LangGraph).
- **Interactive Queries:** We expose simple CLI commands and Streamlit UI (docs / supports / refutes) that walk the graph to answer user questions.

---

3. Handling Conflicting & Low-Confidence Evidence

1. **Minimum Evidence Threshold**
   We require successful ingestion + classification of **at least 3 distinct articles** before emitting a verdict, so a single outlier can't sway the result.
2. **Numeric Averaging**
   We collect all "supports" scores and all "refutes" scores, compute avg_support and avg_refute, and compare them:
   - If avg_support > avg_refute → **Supported**
   - If avg_refute > avg_support → **Refuted**
   - Otherwise → **Inconclusive**
3. **Skip Low-Content Documents**
   Our document_fetcher trims to the first 3–6 paragraphs and enforces a minimum length (skip if len(text) < 100 characters). This prevents empty or boilerplate-only fetches from polluting summaries.
4. **Neutral Classification as No-Op**
   Any snippet classified "neutral" is not added to the graph at all, so truly ambiguous passages don't tilt the averages.

---

4. Technical Challenges

1. **Robust Scraping**
   - **SERP-APIs** sometimes return paywalled or dynamic-rendered pages (empty fetch).
   - ***Solution:***
     1. **newspaper3k** extraction

    2. **BeautifulSoup** fallback to <p> tags
    3. **Paragraph-limit** to ensure concise summaries
2. **Summarizer & Classifier Pipelines**
    - Initial HuggingFace distilbart-cnn + bart-mnli pipelines performed poorly in Docker due to I/O errors and cache permissions and model-loading failures under Slim images.
    - We experimented with mounting a host cache (-v ~/.cache/huggingface:/app/cache/huggingface) and setting HF_HOME, but stability remained brittle.
    - ***Solution:***

      Used Groq llama-3.3-70b API. Switching to Groq llama-3.3-70b API eliminated model-loading brittleness, reduced container size, improved inference speed, and delivered significantly more stable and higher-quality outputs in production.
    -
3. **Context-Length & Token Limits**
    - Multi-article summarization regularly exceeded OpenAI's 4k-token limit, leading to truncated outputs or API errors.
    - ***Solution:***
        - Modified the agent's behavior to **summarize each document individually** instead of batching.
        - Implemented **chunking strategies** where larger documents were split into smaller parts and then summarized separately before re-aggregation.
4. **Docker & Streamlit Watchdog**
    - Streamlit's file-watcher struggled under container volume mounts, triggering "no running event loop" errors. We disabled auto-reloading or switched to mounting only code (not large caches) to work around.
5. **JSON Decoding in Frontend**
    - Occasionally our summarizer outputs weren't valid JSON (e.g. empty strings, stray whitespace), causing json.loads to fail in the Stream Lit logs. We inserted guard clauses and strip() calls around every summarize_and_classify output.

---

5. Possible Next Steps

1. **Persistence & Visualization**
    - Swap in **LangGraph** or a lightweight SQLite backend so graphs survive across sessions.
    - Export to Graphviz/Plotly for visual evidence-maps.
2. **Real-Time Monitoring**
   Schedule recurring ingestion tasks (e.g. "daily scan for new articles on *claim X*") and notify on new supporting/refuting evidence via automations or webhooks.
3. **Unit & Integration Tests**
