Reflection Paper

A. Architecture & Design Decisions

Our project was designed as an end-to-end Retrieval-Augmented Generation (RAG) system that answers climate-related questions using structured historical data. From the outset, our group prioritized modularity, reproducibility, and clear separation of responsibilities, allowing different components of the system to be developed and tested independently before being integrated into a complete pipeline.

On the data side, we selected two global datasets from Our World in Data: annual country-level CO_2 emissions and monthly temperature anomaly records. Because these datasets differed in temporal granularity, we aggregated the monthly temperature anomalies into annual averages and merged both datasets by country and year through a reusable ETL script. This approach avoided ad hoc preprocessing and ensured that the cleaned dataset could be regenerated consistently. While the full dataset covers many countries globally, we used the United States and Canada as representative examples during exploratory analysis and testing to validate trends and schema consistency.

For retrieval, each row of the cleaned dataset was converted into a natural-language text representation suitable for semantic search. We used the `bge-small-en-v1.5` embedding model, which offers a strong balance between semantic quality and computational efficiency in a limited-memory environment. To improve runtime performance, the embedding model was initialized through a cached loader with automatic CPU/GPU selection and embedding normalization, ensuring stable similarity comparisons across queries.

We used FAISS as the vector store and wrapped it behind a minimal interface that exposes only document insertion and similarity search. This abstraction allowed the rest of the pipeline to remain agnostic to the underlying index implementation. The FAISS index was persisted to disk and reloaded across runs, which reduced startup costs and improved reproducibility during development and testing.

Overall, these architectural decisions emphasized clarity, modularity, and ease of integration, enabling each team member to contribute independently without breaking the end-to-end system.

B. Retrieval Quality & Failure Analysis

The retrieval system performs best for queries that directly reference explicit information contained in the constructed documents, such as emissions or temperature anomaly values for a specific country and year. For example, a query like *“What was the temperature*

anomaly in Canada in 2021?" reliably retrieves the corresponding document row with clear numeric evidence.

The retriever enforces a minimum of $k \geq 3$ results and always returns source metadata, including dataset identifiers and readable text snippets. This design improves transparency and allows users to trace each answer back to the underlying data rather than treating the model output as an opaque response.

However, several limitations were observed. One failure mode occurs when relevant evidence is embedded deep within a long, concatenated document. In such cases, the correct document may be retrieved, but the short snippet displayed to the user does not always surface the most relevant supporting sentence. This highlights a trade-off between document length and evidence visibility.

Another failure case arises when users ask about variables that are not present in the dataset at all. In these situations, the system correctly refuses to hallucinate and instead responds that it does not have enough information to answer the question. While this behavior may appear restrictive, it reflects proper grounding and aligns with the project's goal of faithful retrieval over speculative generation.

C. API & Engineering Challenges

One of the most challenging aspects of the project was transforming a working RAG pipeline into a reliable, testable Flask API that could be run by others with minimal setup. While individual components—ETL processing, embeddings, FAISS indexing, and retrieval—worked correctly in isolation, integrating them into a persistent API introduced several non-obvious engineering challenges.

A major issue involved pipeline initialization and performance. In an early implementation, the RAG pipeline was constructed inside the API request handler, causing the embedding model and FAISS index to reload on every request. This significantly increased latency and risked exceeding memory limits on the virtual machine. I resolved this by implementing a lazy-loading approach, caching the pipeline instance at the module level so it is built once and reused across requests.

Debugging was another challenge. Initially, the API returned only a generic "Internal error" message when failures occurred. By temporarily enabling full tracebacks during development, I was able to identify issues such as missing FAISS index files and embedding model loading errors. After resolving these problems, I restored cleaner error handling to keep API responses concise and user-friendly.

I also encountered environment reproducibility issues. Despite installing all dependencies via `requirements.txt`, the VS Code static analyzer reported missing imports due to mismatches between the editor's Python interpreter and the runtime environment. Although the API executed correctly in the terminal, this experience reinforced the importance of

validating systems through actual execution rather than relying solely on static analysis warnings.

D. Team Collaboration & Process

Our team followed a clearly defined division of responsibilities that mirrored a realistic data systems workflow, with each member owning a distinct stage of the pipeline while coordinating closely at integration points.

Keer served as the **Data Ingestion Lead**, designing and implementing the ETL pipeline that merged global climate datasets, aggregated monthly temperature anomalies into annual values, and aligned them with country-level CO_2 emissions. Keer also converted cleaned tabular data into natural-language text suitable for embedding and managed the organization and validation of the data/ directory.

Zhouyue acted as the **Vector Store & Retrieval Lead**, implementing the embedding pipeline using the BAAI bge-small-en-v1.5 model and building the FAISS vector store. He ensured reliable similarity search with a minimum top-k of three results, preserved metadata across retrieval, and implemented index saving and loading to avoid repeated recomputation.

Michelle was responsible for **API integration and model orchestration**, packaging the existing RAG components into a clean, testable Flask service. This included implementing the /api/ask and /api/health endpoints, enforcing rubric constraints, and returning structured JSON responses containing both grounded answers and traceable sources.

Our workflow was highly iterative. We first ensured that each component worked independently before integrating them into a complete end-to-end system. GitHub was used for version control with frequent, incremental commits, which simplified debugging and made integration issues easier to track. Through clear ownership, continuous communication, and early integration, our team successfully built a reproducible RAG system that meets the project requirements and can be reliably run and tested by others.