Authematic: Developer Documentation Guide

This document provides a comprehensive technical guide for developers working on the Authematic project. It covers the system architecture, setup procedures, codebase structure, and the detailed workings of the data processing pipeline and interactive web application.

Part 1: Architecture, Setup, and Project Structure

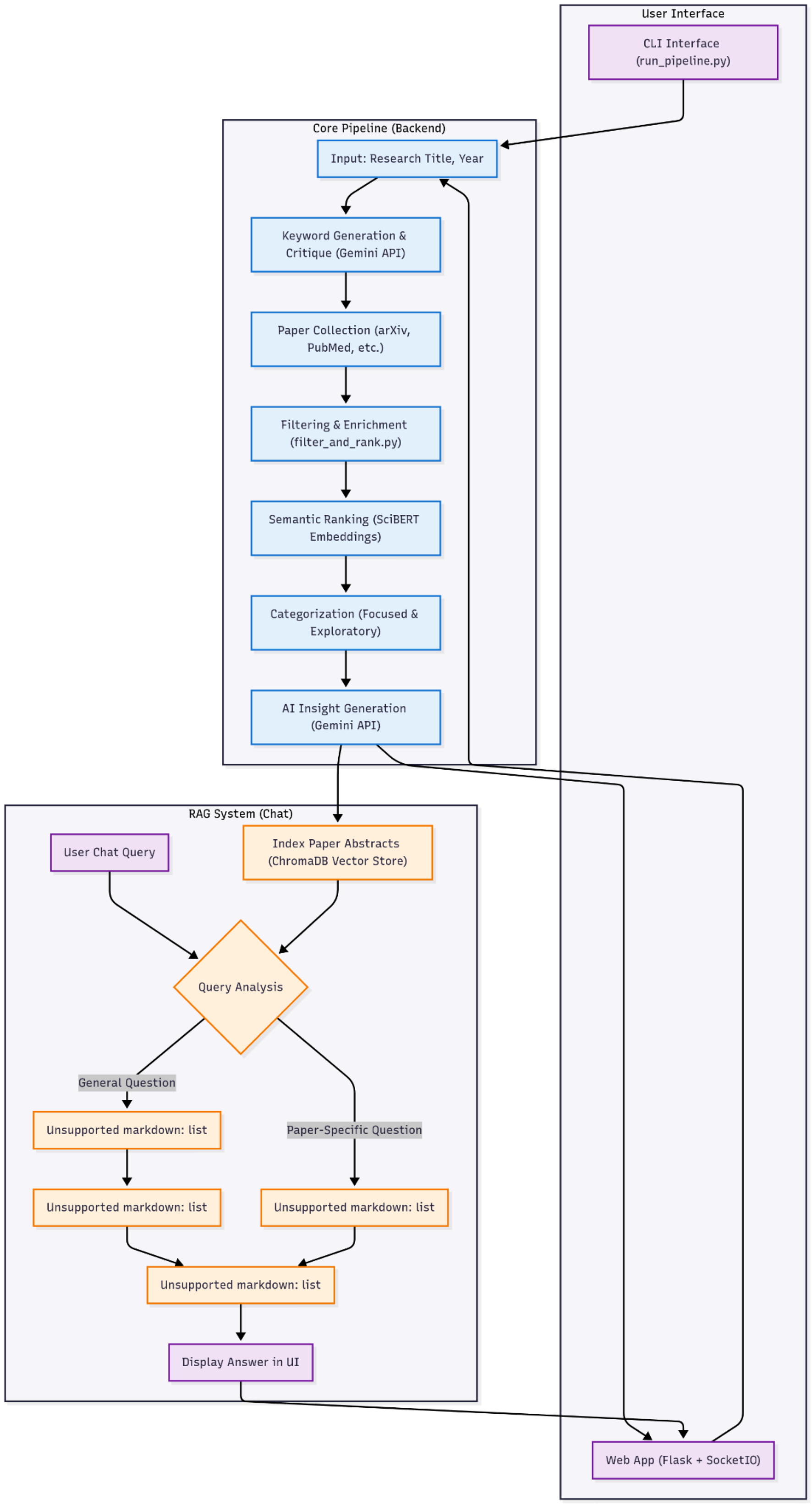
1. High-Level Architecture

Authematic is an AI-powered literature curation assistant. Its primary function is to take a user's research title and generate a curated, ranked, and categorized list of relevant academic papers. The system has evolved to include a sophisticated interactive layer, allowing users to "chat" with the collected body of research.

The system is composed of two main parts:

1. **Core Data Pipeline:** A sequence of Python scripts responsible for the heavy lifting. It takes a research title, generates keywords, scrapes academic sources (arXiv, PubMed, CrossRef), enriches the data, and uses semantic models (SciBERT) to filter, rank, and categorize the findings. This can be run via a Command Line Interface (CLI).
2. **Web Application & RAG Chat Interface:** A Flask and SocketIO-based web application provides an interactive, real-time user experience. The key innovation here is the **Retrieval-Augmented Generation (RAG)** system. After the core pipeline runs, the abstracts of the collected papers are indexed into a **ChromaDB** vector store. This allows developers and end-users to ask natural language questions and receive answers grounded in the context of the discovered literature.

The overall architectural flow is illustrated in the next page:


User Interface
CLI Interface (run_pipeline.py)
Core Pipeline (Backend)
Input: Research Title, Year
Keyword Generation & Critique (Gemini API)
Paper Collection (arXiv, PubMed, etc.)
Filtering & Enrichment (filter_and_rank.py)
Semantic Ranking (SciBERT Embeddings)
Categorization (Focused & Exploratory)
AI Insight Generation (Gemini API)
RAG System (Chat)
User Chat Query
Index Paper Abstracts (ChromaDB Vector Store)
Query Analysis
General Question
Paper-Specific Question
Unsupported markdown: list
Unsupported markdown: list
Unsupported markdown: list
Unsupported markdown: list
Display Answer in UI
Web App (Flask + SocketIO)

Authematic Implementation Diagram

2. Local Development Environment Setup

Follow these steps to set up a local environment for development and testing.

1. Prerequisites:

- Python 3.10+
- git for cloning the repository.

2. Clone the Repository:

```
Bash
git clone https://github.com/GenAIPHBuilders-org/team-Autonominds-2025.git
cd team-Autonominds-2025
```

3. Set up a Python Virtual Environment:

This is crucial for managing project dependencies without affecting your global Python installation.

```
Bash
# For macOS / Linux
python3 -m venv venv
source venv/bin/activate

# For Windows
python -m venv venv
venv\Scripts\activate
```

4. Install Dependencies:

The project's dependencies are listed in web_app/requirements.txt. Install them using pip.

```
Bash
pip install -r web_app/requirements.txt
```

Note: The requirements.txt file includes all necessary packages for both the core pipeline and the web application, such as flask, flask-socketio, google-generativeai, transformers, torch, and chromadb.

5. Configure Environment Variables:

The application uses a .env file to manage API keys. Create a file named .env in the project root directory (team-Autonominds-2025/).

The api_client_manager.py script is responsible for loading this key. Add your Google Gemini API key to the .env file:

Code snippet

```
GEMINI_API_KEY="your_api_key_here"
```

6. Model Downloads (First Run):

The first time you run the application, the transformers library will download the allenai/scibert_scivocab_uncased model, which is used for generating semantic embeddings. This may take a few minutes and requires an internet connection. Subsequent runs will use the cached model.

3. Project Structure

The project is organized into core backend modules and a web_app directory.

```
team-Autonominds-2025/
├── .env                # Stores API keys and other secrets.
├── run_pipeline.py      # 🚀 ENTRY POINT for the Command Line Interface (CLI).
|
├── api_client_manager.py # Manages and configures the Gemini API client.
├── paper_collector.py    # Handles keyword generation and fetching papers from sources.
├── keyword_critic.py     # Refines generated keywords using an LLM critic.
├── filter_and_rank.py    # Filters, ranks, and categorizes papers.
├── embeddings.py         # Generates SciBERT embeddings for text.
├── extract_insights.py   # Generates AI summaries and relevance statements for papers.
├── vector_store_manager.py # Manages the ChromaDB vector store for the RAG system.
|
├── README.md            # The original, user-focused README file.
└── web_app/
    ├── app.py          # 🚀 ENTRY POINT for the Flask web application and SocketIO server.
    ├── requirements.txt  # Python dependencies for the project.
    ├── chroma_db_store/  # Persistent storage for the ChromaDB vector database.
    |
    ├── static/         # Contains frontend assets.
    │   ├── css/style.css # Main stylesheet for the web interface.
    │   └── js/main.js    # Core JavaScript for SocketIO communication and UI logic.
    |
    └── templates/      # Contains Flask HTML templates.
        ├── base.html     # Base template with header, footer, and script imports.
        └── index.html    # Main template for the chat interface.
```

Core Module Descriptions:

- **run_pipeline.py**: Orchestrates the entire data processing pipeline from the command line. It takes user input for the title and year, then calls the other modules in sequence to produce the final ranked lists, printing them to the console.
- **api_client_manager.py**: A simple but crucial module that loads the GEMINI_API_KEY from the .env file and configures the google-generativeai client for use throughout the application.
- **paper_collector.py**: The starting point of the data pipeline. It contains functions to generate topics, sub-themes, and keywords using the Gemini API. It also includes the functions for concurrently fetching paper data from academic sources like arXiv, PubMed, and CrossRef.
- **keyword_critic.py**: Implements an AI-based critique step. It takes a list of generated keywords, asks the Gemini model to act as a critic to remove redundant or broad terms, and suggests better alternatives.
- **filter_and_rank.py**: A central module for data cleaning and ranking. It provides functions to filter papers by checking for valid DOIs and abstracts, deduplicate entries, and, most importantly, perform semantic ranking using SciBERT embeddings.
- **embeddings.py**: A wrapper around the Hugging Face transformers library. Its sole purpose is to provide a consistent function, embed_text, for generating 768-dimensional SciBERT vectors for any given text. It includes an in-memory cache for performance.
- **extract_insights.py**: This module uses the Gemini API to perform analysis on a final paper's abstract. It generates a formal summary and a second-person explanation of the paper's relevance to the user's original research title.
- **vector_store_manager.py**: Manages all interactions with the ChromaDB vector store. It handles indexing paper abstracts (build_rag_index) and retrieving relevant context for user queries (query_rag_index).

Web Application Descriptions:

- **web_app/app.py**: The heart of the interactive application. This script initializes a Flask server and uses Flask-SocketIO to handle real-time, bidirectional communication with the frontend. It manages user sessions, triggers the backend pipeline as a background task, and hosts the RAG chat logic.
- **web_app/static/js/main.js**: This JavaScript file manages all frontend interactivity. It establishes the SocketIO connection, sends user messages to the backend, and listens for events from the server (like receive_message, progress_update, and results_ready) to dynamically update the UI.
- **web_app/templates/index.html**: The main HTML file that structures the user interface. It defines the chat container, input box, and the dynamic areas where results and logs will be displayed.

Part 2: Core Data Pipeline Deep Dive

The core pipeline is the engine of Authematic. It's a sequence of orchestrated steps designed to transform a single research title into two highly relevant, categorized lists of academic papers. The main orchestrator for this flow when run from the command line is **run_pipeline.py**.

The pipeline can be broken down into five distinct phases:

Phase 1: Keyword Generation & Critique

This initial phase is about deconstructing the user's research title into a structured set of high-quality search terms. The primary module responsible for this is **paper_collector.py**.

1. **Topic Generation**: The process starts with the user's research title. The generate_topics(title) function is called, which prompts the Gemini API to return four high-level academic topics that frame the title, including "Core," "Adjacent," and "Emerging" categories.
2. **Sub-theme Generation**: For each of the four topics, the generate_subthemes(related_topics) function is called. This queries the Gemini API again to break down each topic into more specific research niches or sub-themes.
3. **Keyword Generation**: With a structure of Topic -> Sub-theme, the generate_keywords_by_subtheme(...) function is invoked. It prompts the API to generate a list of 2-5 word, high-specificity search keywords for each sub-theme, ensuring the keywords combine the context of both the parent topic and the specific sub-theme.
4. **AI-Powered Critique**: To improve search quality, each list of generated keywords is passed to the critique_list(label, candidates) function in **keyword_critic.py**. This function uses a separate LLM prompt that instructs the AI to act as an "expert academic research critic". It removes overly broad or redundant terms and can suggest more precise replacements, which are then added to the final list of keywords.

Output of Phase 1: A clean, nested dictionary of Topic -> Sub-theme -> [List of critiqued keywords].

Phase 2: Paper Collection & Enrichment

Using the refined keywords, this phase fetches paper candidates from multiple academic sources. This is orchestrated by the collect_papers(...) function in **paper_collector.py**.

1. **Concurrent Fetching**: The system iterates through each sub-theme's keyword list. For each keyword, it uses a ThreadPoolExecutor to query multiple literature sources in parallel. The search functions include search_arxiv, search_pubmed, and search_crossref.

2. **Bucket-Based Aggregation**: As papers are fetched, they are added to a "bucket" corresponding to their sub-theme. The collection process for a sub-theme continues across keywords and fetch attempts (offsets/pages) until a minimum number of papers (e.g., 35) is gathered in the bucket. This ensures a healthy pool of candidates for each research angle.
3. **Initial Filtering**: During collection, papers are immediately filtered by the cutoff_year, and a basic check is performed to ensure a DOI and abstract are present.
4. **Data Enrichment**: After the initial collection, the enrich_with_semantic_scholar(papers) function is called. This function iterates through the collected papers and, for any that are missing an abstract or author data, it attempts to fetch the missing details from the Semantic Scholar API using the paper's DOI. This significantly improves the quality and completeness of the data for the ranking phase.

Output of Phase 2: A single, enriched, but not yet fully filtered, list of all collected paper dictionaries.

Phase 3: Thematic Term Generation

Parallel to paper collection, the system generates three sets of "thematic" terms from the original title. These terms are used later for boosting scores and categorizing papers. This logic is found in **paper_collector.py** and **run_pipeline.py**.

1. **Generate Raw Terms**: Three functions in paper_collector.py call the Gemini API with specific prompts:
    - generate_domain_terms(title): To get core subject matter keywords.
    - generate_app_terms(title): To get concrete application phrases (the "what" and "where").
    - generate_tech_terms(title): To get methodology or technique phrases (the "how").
2. **Critique and Clean**: Just like the search keywords, these lists are refined using critique_list from **keyword_critic.py**. They then go through a final cleaning step in run_pipeline.py using the clean_terms(terms) helper function, which removes generic methodology stop-words (like "framework", "analysis", "method").

Output of Phase 3: Three refined lists: final_app_terms, final_tech_terms, and final_domain_terms.

Phase 4: Semantic Ranking & Score Boosting

This is the core ranking phase where raw relevance is calculated and then adjusted based on thematic fit.

1. **Final Pre-Rank Filtering**: The collected papers are first passed through a final, stricter set of filters from **filter_and_rank.py**: filter_by_doi, filter_by_abstract, and dedupe_by_doi. This ensures only high-quality, unique papers proceed to the expensive ranking step.

2. **Semantic Vectorization**: The semantic_rank_papers(query, papers) function in **filter_and_rank.py** is called. It gets a 768-dimensional vector embedding for the original query_title and for the title/abstract of each paper by calling the embed_text(text) function from **embeddings.py**. This embedding is powered by the allenai/scibert_scivocab_uncased model.
3. **Cosine Similarity**: A raw score is calculated for each paper based on the cosine similarity between its vector and the query's vector. This score represents pure semantic relevance.
4. **Thematic Boosting**: The logic in run_pipeline.py then iterates through the ranked papers. It checks if a paper's text matches the final_app_terms and final_tech_terms generated in Phase 3.
    - If a paper matches **both** an application and a technique term, its score is boosted by a factor of **1.25**.
    - If it matches only one, it receives a smaller boost of **1.10**.
5. **Re-sorting**: The papers are sorted again based on this final, boosted score.

Output of Phase 4: A single list of all valid papers, sorted by a boosted score that reflects both semantic and thematic relevance.

Phase 5: Final Categorization

The final step, handled in **run_pipeline.py**, is to divide the ranked papers into the two deliverable lists.

1. **Focused Papers**: These are the most relevant papers. The selection uses a tiered filtering logic:
    - **Tier 1 (Strictest)**: The system first looks for papers that match at least one application term, one technique term, AND one domain term.
    - **Fallback Tiers**: If not enough papers are found, it broadens the criteria, for example, looking for papers that match just an application AND a technique term. This ensures the list is populated with the best available candidates.
2. **Exploratory Papers**: These papers provide broader context. They are selected from the remaining papers not chosen for the "Focused" list. The criteria are looser, typically requiring matches with domain terms to find papers that are related but may use different techniques or study different applications.

Output of Phase 5: Two final lists, final_focused_papers (top 20) and final_exploratory_papers (top 10), ready to be presented to the user.

Part 3: Web Application & RAG Chat Deep Dive

This part of the guide moves from the backend data processing pipeline to the interactive user-facing components. The web application, located in the web_app/ directory, not only serves as a user interface but also orchestrates the pipeline and enables the advanced RAG chat features.

3.1. Backend: Flask & SocketIO Server (web_app/app.py)

The web server is built using Flask, a lightweight Python web framework, and extended with Flask-SocketIO for real-time capabilities.

- **Flask's Role**: Flask is used for the basic web server responsibilities. It serves the main index.html page and handles standard HTTP requests, such as the /health check route used for testing deployment health.

- **SocketIO for Real-Time Communication**: The core of the interactive experience is enabled by Flask-SocketIO. It allows for bidirectional, event-based communication between the Python backend and the JavaScript frontend. This is essential for:

    - **Interactive Chat**: Sending and receiving messages without needing to reload the page.
    - **Live Progress Updates**: The server can push progress_update events to the user during the long-running pipeline, keeping them informed of the status.
    - **Live Log Streaming**: Standard output from the backend pipeline is captured and streamed to the UI via the server_log event, providing a detailed, real-time log for developers or advanced users.
- **Handling Long-Running Processes**: The paper collection pipeline can take several minutes to complete. To prevent the web server from becoming unresponsive, the entire pipeline is initiated as a background task using socketio.start_background_task(run_pipeline, ...). This allows the server to remain available to handle other requests while the pipeline works in the background.

- **Session Management**: The application uses a simple, in-memory Python dictionary named USER_SESSIONS to manage user state. The user's unique request.sid (SocketIO session ID) is used as the key. This dictionary stores the final list of papers and the unique name of the ChromaDB collection created for that user's RAG session. *Note: For a production environment, this should be replaced by a more robust solution like Redis or a database.*

3.2. Frontend: UI Logic (web_app/static/js/main.js)

All client-side interactivity is managed by **main.js**.

- **Connection & Event Handling**: The script establishes a Socket.IO connection to the server. It then defines handlers for emitting events (sending data to the server) and listening for events (receiving data from the server).

- **Client-to-Server Events (Emitters)**:

  - send_message: Fired when the user sends a message in the initial chat interface. It carries the message text and the current chatContext (e.g., the research title).
  - sidebar_chat_message: Fired when the user sends a message *after* the results are displayed in the tabbed interface. It includes the query and a context object specifying if it's a "general" chat or a "specific" paper chat.
- **Server-to-Client Events (Listeners)**:

  - receive_message & progress_update: These events append new messages from the bot to the main chat window.
  - results_ready: This is the most important handler. When the pipeline is finished, the server emits this event with the final list of papers. The JavaScript code then dynamically dismantles the initial UI and builds the tabbed results interface, creating a "General Chat" tab and collapsible lists for "Focused" and "Exploratory" papers.
  - sidebar_chat_response: After a RAG chat query is processed, this event delivers the answer, which is then appended to the chat history of the currently active tab.

3.3. Retrieval-Augmented Generation (RAG) System

The RAG system is what allows users to "chat with their results." It ensures that the LLM's answers are grounded in the content of the discovered papers, not its own general knowledge.

1. Indexing the Research (vector_store_manager.py)

1. **Triggering**: After the core pipeline produces the final lists of focused and exploratory papers, the run_pipeline function calls build_rag_index from **vector_store_manager.py**.
2. **Collection Creation**: A unique collection name is generated for the session, and a ChromaDB persistent client is used to create or get this collection.
3. **Embedding and Upserting**: The build_rag_index function iterates through the final papers.
   - It takes the abstract of each paper.
   - Crucially, it manually generates a SciBERT embedding for the abstract by calling embed_text(abstract) from **embeddings.py**.
   - It then upserts the data into ChromaDB, providing the generated embeddings, the documents (the abstract text itself), and metadatas (like title and DOI).

2. Querying for General Chat (web_app/app.py)

When the user asks a question in the "General Chat" tab, the get_general_rag_response function is executed.

1. **Embed User Query**: The user's text query is converted into a SciBERT vector using embed_text(query).
2. **Retrieve Relevant Context**: This query vector is used to search the session's ChromaDB collection (collection.query). The function retrieves the top 5 most semantically similar document chunks from the indexed abstracts. These chunks form the "context" for the LLM.
3. **Generate Grounded Answer**: A detailed prompt is constructed. This prompt contains strict instructions for the Gemini model, the retrieved context chunks, and the original user question. The prompt guides the model to answer *only* based on the provided text and to adopt different response modes (e.g., Analyze, Compare, Summarize) as needed.

3. Querying for Single-Paper Chat (web_app/app.py)

This is a simplified version of RAG handled by the get_single_paper_response function.

1. **Identify the Paper**: The system identifies which paper the user is currently viewing based on the active tab's DOI.
2. **Direct Context**: Instead of a vector search, the "context" is simply the full title and abstract of that single paper.
3. **Generate Answer**: A simpler prompt is sent to the Gemini model, containing this direct context and the user's question, asking it to answer based only on the provided information.

Part 4: Coding Standards, Testing, and Deployment

This final section outlines the best practices and conventions observed in the Authematic codebase. Adhering to these standards will make it easier for new developers to contribute and will ensure the long-term health of the project.

4.1. Coding Standards & Conventions

The codebase follows standard Python conventions and a modular design philosophy.

- **Code Style**: The Python code largely adheres to the **PEP 8** style guide. New contributions should follow this standard for variable naming, line length, and spacing.

- **Type Hinting**: The backend code makes extensive use of Python's type hints (e.g., List, Dict, Optional, str). This is a critical convention in this project. All new functions and methods should include type hints for their parameters and return values to improve code clarity and allow for static analysis.

- Python

# Example from filter_and_rank.py

```
def filter_by_doi(papers: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

    # ... function body ...
```

-
- **Docstrings & Comments**: Functions are documented using clear docstrings that explain their purpose, arguments (Args:), and return values (Returns:), as seen in modules like filter_and_rank.py. Complex or non-obvious lines of code are explained with inline comments.

- **Variable Naming**:

    - snake_case is used for variables and function names (e.g., collect_papers, final_app_terms).
    - UPPER_SNAKE_CASE is used for constants (e.g., METHODOLOGY_STOP_TERMS).
- **Modularity**: The project is well-structured into separate modules, each with a distinct responsibility (e.g., embeddings.py only handles vectorization, vector_store_manager.py only handles ChromaDB interactions). This separation of concerns should be maintained. New, distinct functionalities should be placed in their own modules.

4.2. Testing Guide

The project includes several built-in methods for testing and health checks.

- **Existing Module Tests**: The **filter_and_rank.py** module contains several internal test functions, such as _test_doi_filter() and _test_semantic_rank(). These can be executed by running the script directly from the command line:

- Bash

```
python filter_and_rank.py
```

-
- This will run a series of assertions against sample data to verify the functionality of the filtering and ranking logic.

- **Web Application Health Checks**: The web application in **web_app/app.py** includes two useful routes for quick, manual testing:

    - /health: Returns a simple JSON response confirming that the Flask server is running.
    - /test-gemini: A route that makes a simple test call to the Gemini API. This is extremely useful for verifying that the GEMINI_API_KEY is configured correctly and that the service is reachable.

Recommended Future Testing Strategy

To improve test coverage and reliability, the following strategy is recommended:

1. **Adopt a Framework**: Use a formal testing framework like pytest.
2. **Create a tests/ Directory**: All new test files should be placed in a dedicated tests/ directory at the project root.
3. **Mock External APIs**: For unit tests, it's crucial to "mock" external API calls (to Gemini, arXiv, PubMed, etc.). This makes tests:
    - **Fast**: They don't have to wait for slow network responses.
    - **Reliable**: They won't fail if an external service is down.
    - **Free**: They won't consume API quotas. Python's built-in unittest.mock library is well-suited for this.

4.3. Deployment & Troubleshooting

Local Development Server (For Testing & Debugging)

For local development, you should use the built-in development server provided by Flask and Flask-SocketIO. This server is easy to start and provides useful features like hot-reloading and an interactive debugger.

1. **Navigate to the project root directory.**
2. **Run the app.py script directly**:
3. Bash

```
python web_app/app.py
```

4.
5. This command executes the if __name__ == '__main__': block in the script, which calls socketio.run(app, debug=True). The debug=True argument enables the features mentioned above.
6. Open your browser to http://127.0.0.1:5000 to interact with the application.

Important: This development server is **not suitable for production use**. It is not designed to be as stable, secure, or performant as a production-grade WSGI server.

Production Deployment (Gunicorn)

For a live deployment, the web application should be run with a production-ready WSGI server. The requirements.txt file already includes gunicorn, a popular choice for this purpose.

To run the application for production:

1. **Navigate to the project root.**
2. **Run Gunicorn**: Point it to the Flask application instance (app) inside the web_app.app module.
3. Bash

```
# Binds the server to all network interfaces on port 5000

gunicorn --bind 0.0.0.0:5000 web_app.app:app
```

4.
5. **Environment Variables**: Ensure that any required environment variables (especially GEMINI_API_KEY) are set and available in the shell or service where you run the Gunicorn command.

Troubleshooting Common Issues

- **API Key Errors**: If the application fails with authentication errors, first check the /test-gemini endpoint. The most common cause is a missing or incorrect GEMINI_API_KEY in your .env file or deployment environment variables.
- **Model Downloads on First Run**: The embeddings.py module needs to download the SciBERT model from Hugging Face on its first execution. If the application fails on startup, ensure the server has an active internet connection. Subsequent runs will use the cached model.
- **ChromaDB Errors**: Errors related to the RAG system or vector_store_manager.py may be due to file permissions. Ensure that the application has the necessary rights to create and write to the web_app/chroma_db_store/ directory.
- **Dependency Conflicts**: Always use the provided web_app/requirements.txt file within a dedicated Python virtual environment to avoid issues with package versions.