System Design

1. High-level Design

The system is a monolithic application containing a frontend web interface, a backend API service, and data storage, focused on searching, analyzing, and displaying arXiv papers.

- **Frontend**: Built using HTML, Tailwind CSS, JavaScript (jQuery, ECharts), providing a user interface for searching, browsing arXiv papers, and viewing analysis results (sentiment, clustering, metadata). Includes `arxiv.html` and `arxiv_analysis.html`.
- **Backend API**: Built using Flask (Python), handling frontend requests, invoking business logic, and interacting with the database and the arXiv API. Includes `app.py`, `arxiv_client.py`, `sentiment_analyzer.py`, `paper_clustering.py`.
- **Data Storage**: Uses an SQLite database (`data/test_db.sqlite`) to store cached data for arXiv papers (`arxiv_papers` table). Relies on the local filesystem to store the arXiv metadata snapshot.
- **External Services**: arXiv API, used for real-time searching and fetching paper data.
- **Core Flows**:
    i. **arXiv Search (`/arxiv/search` in `app.py -> arxiv_client.py`)**: User initiates a search via `arxiv.html` -> Flask backend calls `arxiv_client` -> `arxiv_client` queries the arXiv API -> Returns results to the frontend -> Optionally performs sentiment analysis (`sentiment_analyzer`) and clustering (`paper_clusterer`) -> Stores results in SQLite (`arxiv_papers` table).
    ii. **Unified Search (`/unified_search` in `app.py`)**: User searches (primarily via `arxiv.html`) -> Flask backend calls `arxiv_client` (using `query` or inferred `author_name` if `entity_type` =='person') to search arXiv -> Performs sentiment analysis on results -> Stores results in `arxiv_papers` table -> Returns arXiv results.
    iii. **Sentiment Analysis (integrated in `/arxiv/search` , `/unified_search`)**: Calls `sentiment_analyzer` to analyze arXiv paper abstracts.
    iv. **Clustering (integrated in `/arxiv/search` , `/unified_search`)**: Calls `paper_clusterer` to cluster arXiv paper abstracts.
    v. **Metadata Analysis (`/arxiv/metadata/analysis -> arxiv_analysis.html`)**: Reads the local arXiv metadata snapshot file, performs statistical analysis, and displays it on the frontend using ECharts.

2. Detailed Design - Prototype Components

- **`app.py - /unified_search` Endpoint**:
  - **Input**: `search_type` (arxiv, all), `query` (arXiv query), `entity_type` ('person' for author search), `entity_name` (used as author name if type is 'person'), `start`, `max_results`, `cluster`, `n_clusters`.
  - **Processing**:
    - Parse parameters.
    - **arXiv Search**:
      - Call `arxiv_client.py`'s `search_papers` or `search_by_author` based on `query` or (`entity_type` =='person' and `entity_name`).
      - Fetch arXiv results.
      - Perform sentiment analysis on the results.
      - Use `arxiv_client.save_papers_to_db` to store results in the `arxiv_papers` table.
    - **Clustering**: If `should_cluster` is true, extract abstracts from arXiv results and call `paper_clusterer`.
  - **Output**: JSON object containing `arxiv_results`, `clusters` (if requested).
- **`arxiv_client.py - search_by_author`**:
  - **Input**: `author_name`, `start`, `max_results`.
  - **Processing**:
    - Attempt an exact query using `search_papers(search_type='author')` (e.g., `"John Doe"`).
    - If failed and the name contains spaces, try the `Last, First` format (e.g., `"Doe, John"`).
    - If failed and the name contains `.`, try a simplified query (e.g., `"J Doe"`).
    - If all fail, fall back to `search_papers(search_type='keyword')`.
    - Merge results from all attempts, deduplicate based on `arxiv_id`, sort by `relevance_score`.
  - **Output**: Dictionary containing the `papers` list and `total_results`.

3. Linking Design to Requirements

- **Search Functionality (arXiv)**: Implemented by `/arxiv/search`, `/unified_search` endpoints in `app.py`, calling `arxiv_client.py`. Frontend interface provided by `arxiv.html`.
- **Paper Detail View**: Supported by `/arxiv/paper/<id>` (first checks DB, then API) via `arxiv_client.py`, endpoint provided by `app.py`, implemented as a modal in `arxiv.html`.

- **Sentiment Analysis**: Core logic in `sentiment_analyzer.py`, integrated into search results (`/arxiv/search`, `/unified_search`) in `app.py`.
- **Clustering Analysis**: Core logic in `paper_clustering.py`, integrated into search results (`/arxiv/search`, `/unified_search`) in `app.py`. Frontend displays cluster chart in `arxiv.html`.
- **Metadata Analysis**: `/arxiv/metadata/analysis` endpoint in `app.py` reads the snapshot file for analysis, frontend displays via `arxiv_analysis.html` and ECharts.
- **Data Persistence**: SQLite database (`arxiv_papers` table) caches arXiv data, local filesystem stores metadata snapshot.

4. Requirements

- **Core Features**:
  - P0: Ability to search arXiv papers by keyword, author, title.
  - P0: Ability to display search results list including title, authors, abstract, categories, sentiment label.
  - P0: Ability to view detailed information for a single arXiv paper.
  - P1: Ability to cluster search results and display visualization.
  - P1: Ability to provide an arXiv metadata analysis dashboard.
- **Non-Functional**:
  - P1: User-friendly interface, easy to use (Usability).
  - P1: Reasonable arXiv search response time (considering API rate limits).
  - P2: System extensibility for future addition of new analysis features.
- **Methodology**: Agile development, prioritizing core arXiv search and display features (P0), followed by iterative addition of clustering, sentiment analysis, and metadata analysis (P1).

5. Language and Technology Choices

- **Backend**: Python + Flask
  - **Rationale**: Rich Python ecosystem with libraries for NLP (Spacy, NLTK), Web frameworks (Flask). Flask is lightweight.
  - **Alternatives**: Django, FastAPI, Node.js.
- **Frontend**: HTML + Tailwind CSS + JavaScript (jQuery, ECharts)
  - **Rationale**: Tailwind CSS enables rapid modern UI development. jQuery simplifies DOM manipulation and AJAX. ECharts is powerful for data visualization.
  - **Alternatives**: Bootstrap, React/Vue/Angular, Chart.js/D3.js.
- **Database**: SQLite

    - **Rationale**: Lightweight, no separate DB server needed, suitable for caching and prototypes, built-in Python support.
    - **Alternatives**: PostgreSQL/MySQL, MongoDB.
- **NLP**: Spacy (for potential future expansion), NLTK (VADER for sentiment analysis)
    - **Rationale**: NLTK VADER is simple, easy to use, specialized for sentiment analysis of text like social media.
    - **Alternatives**: Transformers (Hugging Face), TextBlob.

6. Data Model

SQLite Database (`data/test_db.sqlite`)

1. **`arxiv_papers` Table (arXiv Paper Cache)**
    - `arxiv_id` (TEXT, PRIMARY KEY): arXiv ID (e.g., '2301.12345v1').
    - `title` (TEXT, NOT NULL): Paper title.
    - `abstract` (TEXT): Abstract.
    - `authors` (TEXT): List of authors (JSON string).
    - `categories` (TEXT): List of categories (JSON string).
    - `published` (TEXT): Publication date.
    - `updated` (TEXT): Update date.
    - `pdf_url` (TEXT): PDF link.
    - `relevance_score` (REAL, DEFAULT 1.0): Search relevance score.
    - `json_data` (TEXT): Complete paper metadata (JSON string).

File System

- `arxiv/arxiv-metadata-oai-snapshot.json` : arXiv metadata snapshot file.

Test Plan for Components

Target Component: `sentiment_analyzer.py`

Testing Goal: Verify the correctness, robustness, and performance of the `SentimentAnalyzer` class and its methods (`analyze_text` , `analyze_document` , `get_sentiment_label`). Also, test its integration within `app.py` specifically for arXiv data.

Testing Method: Unit Testing, Integration Testing, Usability Testing (indirectly via API).

Testing Environment: Local development environment, using `pytest` (or another testing framework).

1. Unit Tests (`test_sentiment_analyzer.py`)

Test Class: `TestSentimentAnalyzer`

- **Setup**: Initialize a `SentimentAnalyzer` instance.
- **Test `get_sentiment_label(polarity)`**:
  - `test_label_positive`: Input > 0.3 (e.g., 0.5, 1.0), assert returns 'Positive'.
  - `test_label_negative`: Input < -0.3 (e.g., -0.5, -1.0), assert returns 'Negative'.
  - `test_label_neutral`: Input within [-0.3, 0.3] (e.g., 0.0, 0.2, -0.1), assert returns 'Neutral'.
  - `test_label_boundaries`: Test boundary values 0.3 and -0.3.
- **Test `analyze_text(text)`**:
  - `test_analyze_positive_text`: Input clearly positive text.
  - `test_analyze_negative_text`: Input clearly negative text.
  - `test_analyze_neutral_text`: Input neutral or objective statements.
  - `test_analyze_mixed_text`: Input text with both positive and negative sentences.
  - `test_analyze_empty_text`: Input empty string `""`.
  - `test_analyze_long_text`: Input longer text (simulating paper abstract).
  - `test_analyze_text_structure`: Assert the returned dictionary structure is correct.
  - `test_sentence_splitting`: Check sentence splitting accuracy.
- **Test `analyze_document(doc_path)`**:
  - If kept, test logic remains, but emphasize it might not be the primary usage path anymore.

2. Integration Tests (`test_app_sentiment_integration.py`)

Testing Goal: Verify that `SentimentAnalyzer` is correctly integrated into the Flask application (`app.py`) for arXiv search results.

- **Setup**:
  - Create a Flask test client (`app.test_client()`).
  - May need to mock the return results of `arxiv_client` to control test data.
- **Test Sentiment Analysis Integration in `/arxiv/search`**:

    - `test_arxiv_search_with_sentiment`: Send GET request to `/arxiv/search` (with a valid query), assert status code 200, response JSON contains `papers` list, and each paper object includes `sentiment_label` and `sentiment_score` fields (even if 'N/A').
    - `test_arxiv_search_sentiment_on_no_abstract`: Simulate arXiv returning a paper missing an `abstract`, assert its `sentiment_label` and `sentiment_score` are 'N/A'.
- **Test Sentiment Analysis Integration in `/unified_search`**:
    - `test_unified_search_with_sentiment`: Send GET request to `/unified_search` (with a valid query), assert status code 200, response JSON contains `arxiv_results` list, and each paper object includes `sentiment_label` and `sentiment_score` fields.

3. Usability Testing (Indirectly via API and Frontend)

- **API Usability**:
    - **Consistency**: Check if the structure of sentiment results returned by `/arxiv/search` and `/unified_search` is consistent.
    - **Clarity**: Check if error messages are clear and meaningful.
    - **Performance**: Test API response time.
- **Frontend Display (`arxiv.html`)**:
    - **Label Display**: Check if sentiment labels (Positive, Negative, Neutral) and scores are displayed correctly in search results.
    - **Styling**: Check if background colors/styles for different sentiment labels are applied as expected.
    - **'N/A' Handling**: Check display when abstract is missing or analysis fails.
    - **Loading State**: Ensure loading indicators cover the sentiment analysis time.

Expected Results:

- All unit tests pass.
- All integration tests pass, verifying the API works as expected.
- Sentiment analysis results logically match the nature of the input text.
- The system fails gracefully with meaningful error messages when handling errors.
- The frontend correctly and clearly displays sentiment analysis results.