COMP9900 Info Tech Project

(P14) AI Comment Moderation - RAG and Classification modelling

Project Report

Group: F16A-AVOCADO

Term:T1 2025

Group Member:

Yinglong Cui
Technical Design
(z5497698@ad.unsw.edu.au)

Hang Pan
Technical Design
(z5598515@ad.unsw.edu.au)

Zhongwei Yang
Technical Design
(z5441208@ad.unsw.edu.au)

Xiaotong Zhang
Project Owner
(z5491457@ad.unsw.edu.au)

Jianhong Liu
Technical Design
(z5460416@ad.unsw.edu.au)

Xuanzhi Liu
Scrum Master
(z5455855@ad.unsw.edu.au)

Submission Date: 25/4/2025

Contents

1. Installation Manual

1.1 Initialize Vector Database First

Before starting for the first time, make sure that **you have initialised the vector database**. If any of the **following conditions** are met, be sure to perform the initialisation operation**:**

1. The `backend/chroma_db` directory is empty or doesn't exist
2. You want to use your own custom data for classification

Execute the following command to initialise:

```
cd backend
python setup_vector_db.py
```

If using custom data:

1. Name the custom Excel file data.xlsx and place it in the backend directory.
2. Running preprocessing scripts:

   ```
   python data_preprocessing.py
   ```

3. Place the generated `data.csv` file in the `backend/data` directory.
4. Execute `setup_vector_db.py` again to initialise.

1.2 Prerequisites

3. Python 3.10 + (Backend)
4. Node.js 16 + (Frontend)
5. Docker and Docker Compose (for containerized deployment)
6. OpenAI API Key (since the default API Key is provided for **testing purposes only**, please replace it with your own OpenAI API Key in .env if you need to use it for a long period of time or to process large amounts of data)

1.3 Option 1: Docker Setup (Recommended)

This approach uses Docker to containerize both frontend and backend services.

5. Create a `.env` file in the project root with your OpenAI API key:

   ```
   OPENAI_API_KEY=your_openai_api_key
   ```

6. Make sure you've initialized the vector database as described above:

   ```
   cd backend
   python setup_vector_db.py
   ```

7. Build and start containers (`make sure locate in root folder`):

```
cd ..
pwd
docker-compose up --build -d
```

8. Access the application at http://localhost:3000

1.4 Option 2: Manual Setup

If you prefer to run services directly on your machine:

Backend Setup:

1. Navigate to the backend directory:

```
cd backend
```

2. Create a virtual environment and activate it:

```
python -m venv venv

source venv/bin/activate   # On Windows: venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Create a .env file with required variables:

```
OPENAI_API_KEY=your_openai_api_key
UPLOAD_DIR=./uploads
PORT=8088
```

5. Initialize the vector database:

```
python setup_vector_db.py
```

6. Start the backend server:

```
uvicorn main:app --host 0.0.0.0 --port 8088 --reload
```

Frontend Setup:

1. Navigate to the frontend directory:

```
cd frontend
```

2. Install dependencies:

```
npm install
```

3. Create .env file for API configuration:

```
REACT_APP_API_URL=http://localhost:8088
```

4. Start the development server:

```
npm start
```

5. Access the application at http://localhost:3000

2. System Architecture Diagram

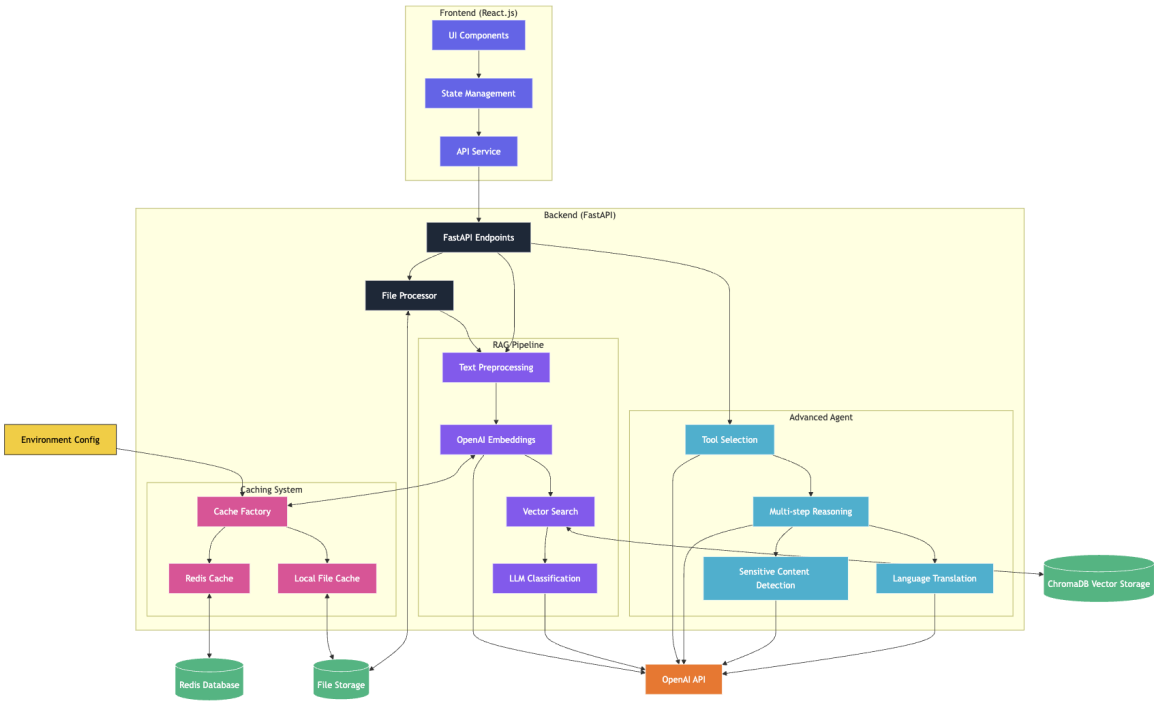

Figure 1. System Architecture Diagram

The system architecture diagram above illustrates the key components of our AI Comment Moderation system. The architecture is divided into four main sections:

1. **Frontend (React):** The user interface layer of the system is provided by this layer. It consists of several key pages for different functionalities and all of them are connected with a central context provider for state management.
   Front-end state management details: This system introduces Zustand state management in the front-end and combines browser LocalStorage to persist key data, which keeps the state continuous even if the page is refreshed or jumped. This design improves the reliability of the system in multi-administrator parallel auditing scenarios: the operation status of each auditor can be synchronised in real-time

without interruption due to interface refresh. In addition, even if the page is repeatedly switched during platform testing or demonstration, the comment data and review progress can be seamlessly connected, which significantly enhances the stability and user experience of the system.

2. **Backend (FastAPI)**: FastAPI is used to build the backend which consists of many API endpoints and core modules:
    - Handle the main functionality, the RAG system for comment classification, the agent system for advanced reasoning, file processing, error handling, caching etc.
    - Interfaces for file upload, comment processing, RAG queries, agent queries, data export.

3. **External Services**: It also integrates with external services:
    - **OpenAI API**: Used for embeddings and LLM inference.
    - **Vector Database (ChromaDB)**: Stores embeddings for efficient similarity search.
    - **Redis Cache (Optional)**: Can be an option for file based caching for faster caching experience in production environments. The system supports file based caching as well as Redis based caching for the high volume scenarios.
4. **Storage**: The system uses local storage for:
    - **File Storage**: Stores uploaded files and processed data.
    - **Embeddings Cache**: Improves performance and reduce number of API calls by caching embeddings. Depending on configuration, this can be file based or Redis based.

The diagram shows the flow of data between the components with arrows. For instance, suppose a user uploads a file via the Data Import Page; in that case, the request will hit the Upload API endpoint, which will then use the File Processor to process the file and will store it in the File Storage. It then passes this data to the RAG or Agent modules where they then work with external services such as OpenAI and the Vector Database to analyze the data.

In particular, our caching system supports two different implementations:

1. **File-based cache**: The default is file based cache where embeddings are stored in the local file system. However, it is easier to set up, but for enterprise scale (high volume) scenarios may have perf issues.
2. **Redis-based cache**: An optional, more scalable solution for production environments. If configured the system will use Redis for faster cache operations with better concurrency support.

The system is designed in a way that each component has a single responsibility and communicates with other components through well defined interfaces. It makes easier the maintenance, the testing and the future extensions of the system.

3. Design Justifications

3.1 Design rationale: optimization and decision-making basis

The design evolution of the key modules in the process of the project development is expatiated and the background, challenges and optimization effect of each adjustment is analyzed in this chapter. Our design is not a static one. It is the result of ongoing trade-offs between real needs, feasibility in terms of technology and user experience. For instance, we gradually optimized data flow and reconstructed the core algorithm to enhance system efficiency. For example, after the initial architecture review we found performance bottlenecks of some modules were not as expected. Scientific decisions are made for these adjustments hence we don't adjust them at random, we base our decisions on the A/B testing, user feedback and the performance monitoring data to ensure that the final solution meets the functional requirements and also has good maintainability and scalability.

3.2 Functional evolution: adjustment from theory to implementation

During the project promotion process, the functional modules have undergone multiple iterations to adapt to the changes in requirements and technical limitations of real scenarios. For example:

Adjustment of algorithm model:
We first adopted a supervised classification model, but in real testing, we found that there was not enough data to have good generalization ability. Upon comparative experiments, we explored a generative approach which not only boosted the performance under small samples but also achieved higher adaptability of the model.

Optimization of database solution:
In the early stage, we planned to make use of a remote vector database for high speed retrieval, but in the deployment stage we found that the operation and maintenance costs were high and highly dependent on the network stability. After evaluation, we changed our usage to an embedded database solution that lowered the deployment complexity, as well as the long term maintenance costs, and provided core functions as needed.

And these are actually reasonable optimizations based on actual operation data, user feedback and engineering practices, which Agile methodology wouldn't allow you to postpone that long, and the end result enables the product to achieve a better balance between function, performance and cost.

3.3 Modular Design Evolution: Iteration and Optimization of Key Architectures

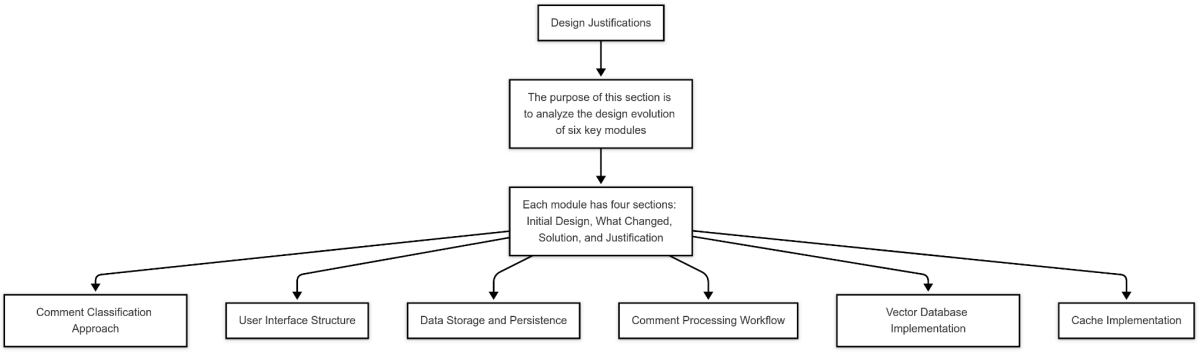

Figure 2. File Storage and Persistence Architecture Diagram

3.4. Design choices and implementation paths for key functional modules

3.4.1. Comment Classification Approach

Initial Design

Initially, we were asked to use the Retrieval-Augmented Generation (RAG) mechanism for comment categorisation at the start of the project. RAG was therefore used as the core classification strategy during the initial design phase. From the start, the team planned a classification process based on the combination of vector retrieval and large-scale language modelling instead of switching to a traditional approach. To address the complex and changing needs of classifying student comments, the initial solution was to build a library of comment semantic vectors and to retrieve examples of similar comments to help the model make classification judgements on new comments.

Reason for design change

In the development process, the RAG classification scheme was kept intact throughout the system architecture without any fundamental changes. The team optimised the RAG process based on test feedback for the implementation details. For instance, the team first considered a third party hosted vector database service for the use case, but eventually switched to using a local embedded vector database to make the deployment easier and also tuned the vector retrieval parameters to enhance the efficiency and accuracy of the similar comment recall. These are not shifts in strategy, but adjustments to the RAG thinking that has been established already, to make sure that the system performance is on expectation.

Solution

The final implementation of the comment classification process fully embodies the RAG idea: the system first generates semantic vectors from new comments via a pre-trained

embedding model (e.g., the text embedding API provided by OpenAI), and retrieves a number of semantically similar historical comments in the built-in vector database. Subsequently, the system submits the retrieved similar comments as reference contexts along with the comments to be classified to a large language model (e.g., GPT-3.5-Turbo) to generate classification results. The model combines the content of the new comment and the reference instance to produce the category label (e.g., 'Complaint' or 'Mantel Health', etc.) and the confidence level of the comment. The whole process does not require pre-labelled large-scale training data, and relies entirely on the existing comment knowledge base and the reasoning capability of the language model to achieve intelligent classification of new comments. For cases where the model is not confident enough, the system will flag the comment for manual review, thus ensuring automation efficiency while taking into account the accuracy and credibility of the results.

```
async def process_comment_with_rag(comment_text: str, vector_store,
llm_chain) -> Tuple:
    similar_comments = await find_similar_comments(comment_text,
vector_store)
    classification = await classify_comment_with_llm(comment_text,
similar_comments, llm_chain)
    return similar_comments, classification
```

1. Find similar comments: The system will recall similar comments that have been processed.

2. Comprehensive judgment: Refer to these similar cases and make classification decisions based on the specific content of the comment

Design reason

The reason for adopting the RAG classification scheme from the beginning to the end is based on the following considerations: (1) Adaptability: RAG uses semantic search to expand the knowledge scope of the model, which can cope with a variety of novel or complex comments without being restricted to a fixed set of predefined labels or training corpus. (2) No need for supervised training: the method eliminates the need for manual labelling of data and training of classification models, and the task can be accomplished by using the existing comment cases and the generic large model, which greatly reduces the development and maintenance costs. (3) Interpretable results: Each classification decision is based on evidence, and the model will refer to similar historical comments to give the basis for judgement, so admins can intuitively understand which cases the system refers to, which helps to improve the trust and transparency of decision-making. (4) Continuous self-improvement: as the system accumulates more comments and their classification results, the semantic indexes in the vector database will become richer and richer, which is equivalent to the accumulation of experience, and the classification effect will be gradually improved. (5) Human-computer collaboration: When the automatic classification is uncertain, the introduction of manual

review is equivalent to letting the AI ‘consult’ human beings, so as to ensure that the final classification is accurate and reliable. These advantages fit the special needs of educational scenarios and ensure the robustness and reliability of the system while improving classification intelligence.

3.4.2. User Interface Structure Optimization

Initial Design

According to the initial Proposal and early Storyboard, the system interface was planned as multiple functional pages, corresponding to data import, automatic review, manual review, and result export. In other words, from the very beginning of the design process, the team adopted a modular layout of multiple pages to reflect the step-by-step process of review processing, and did not intend to stack all the functions into a single page. This structure was already in place at the proposal stage: for example, a separate ‘Import Data’ page was set up for uploading comment data, an ‘AI Agent (Beta)’ page for displaying model classification results and subsequent manual review, and an ‘Export’ page for downloading the processed comments. The ‘Export’ page is used to download the processed dataset, etc., ensuring a clear and organised flow of the interface.

Reasons for the design change

Later in the development and testing process, the multi-page structure was not changed, only the main changes were in the unity of the interface style and optimization of the functional entrances. In the late stage of the project, the global UI theme was improved firstly: the relatively plain interface style in Proposal was upgraded to a darker technological theme to achieve a professional visual outlook. Secondly, a new ‘manual input’ portal was added to the data import page, where users can enter a small amount of comments directly to the front end for experience. This change permits the first time users to enter a few comments manually (without a data file ready) to give a quick taste of auto-categorisation. Last but not least, a ‘AI Agent (Beta)’ section has been added as an experimental page that users can interact with to let the system’s Intelligent Agents run in depth analysis of complex comments. These enrichments of the original design have not modified the basic architecture of separating the modules.

Solution

The adjusted and optimised front-end interface covers a number of functional pages, forming a clear and orderly user operation flow. After entering the system, the user first uses the data import page to upload a comment data file (supporting CSV and other formats), or enters a small amount of text through the new manual input area for testing. Once the data is submitted, the system automatically performs comment analysis and presents the results on an automated review page, which lists the machine-determined category and confidence level for each comment, and highlights entries that require manual review. Next, the user enters the manual review page, where the flagged reviews are manually checked and adjusted, with related similar reviews displayed on the interface for reference to aid decision-making. After

the review is confirmed, the user can go to the data export page to export a file containing the final classification results and review information for archiving or further analysis with one click. In addition to the main process pages, the system also provides the Intelligent Agent (Beta) page as an extension: on this page, users can enter free text queries to trigger the back-end intelligent assistant to conduct complex quizzes and analyses on the review data, e.g., to summarise the opinions based on the context and to detect the implied emotions. This module provides users with an interactive exploration experience with the comment data with the help of a large-scale language model. The entire UI adopts a uniform dark-coloured technological theme, which maintains the consistency of the visual style of each page and the smoothness of the operation experience; users complete the task step by step in the order of importing→automatic reviewing→manual reviewing→exporting, and the functional modules are clearly segregated and interconnected, which is fully in line with the habit of the comment review process.

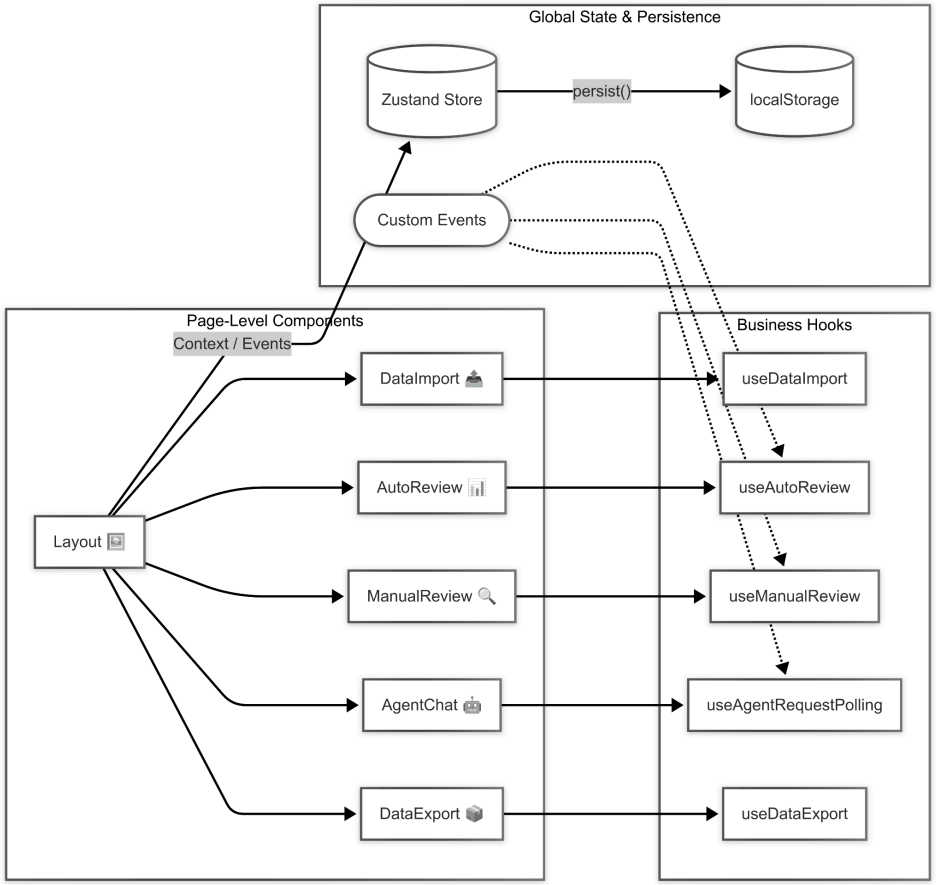

Figure 3. Frontend Global State and Page Modules Architecture

Design reason

The linear flow of reviewing comments makes the multi page interface architecture a natural fit, and makes the interface easier to use and more clear. Step by step interface guidance is easier to use than a single page design that summarizes all functions at once because it allows users to complete tasks by following a given order and prevents the user from being overwhelmed by too much information at the same time. For instance, import, review, and export can be placed on different pages that will allow users to concentrate on the current step

of the operation and minimize cognitive load. Across the board, the use of dark technological theme helped to achieve the professionalism: uniform colour scheme and style of the interface not only make the interface better looking, but also make the interface look rigorous and modern, which fits the positioning of the educational technology application. To lower the barrier to entry, there is a new manual input function on the data import page: first time users can access the system without having to prepare data files in advance, a user friendly design which helps to attract users and quickly demonstrate the value of the system. Regarding the stand-alone Intelligent Agent module, it was designed to provide advanced users with the potential to experience smarter comment analysis service beyond the ordinary categorisation process. This module shows the system is extensible and innovative and provides ideas to enrich the application in the future (e.g., handle more sophisticated text analysis tasks). In brief, all the decisions in the user interface structure design are aimed at improving user experience and meeting requirements, which not only make the operation process clear and convenient, but also bring the system with richer interaction and analysis capabilities.

3.4.3 Data Storage and Persistence

Initial Design

We initially planned a complete database architecture. We originally planned to use traditional database management systems (such as PostgreSQL or SQLite) for data storage and persistence. We hope to achieve structured data access, efficient query functions and good scalability by building standard user-comment-classification data tables to support the diverse functional requirements of subsequent systems. It's like designing a standard library management system for a school library. We plan to use PostgreSQL to establish a standardized data table structure:

- User table (teacher/student information)
- Comment table (original content + metadata)
- Classification result table (label + timestamp)
- Relationship table (association between entities)

This design expects to achieve accurate and efficient data retrieval like library index cards, leaving sufficient space for possible future functional expansion.

Reasons for design change

During the actual development process, we encountered several key problems. First, we realized that although the introduction of traditional databases can improve data management capabilities to a certain extent, it also inevitably increases the complexity of deployment and configuration. This complexity is not only reflected in the initial construction, but also includes a series of operation and maintenance work such as later maintenance, upgrades, backup and disaster recovery, which significantly increases the overall maintenance cost of the system.

Secondly, from the perspective of project requirements, the current functional implementation has low requirements for data consistency and transactionality, mainly based on simple read and write operations, and does not involve complex cross-table transactions or high-concurrency write scenarios. Therefore, the "heavy architecture" brought by traditional database systems (such as relational databases) - such as connection pool management, transaction isolation level tuning, data migration strategy design, etc. - for this project, it constitutes an unnecessary technical burden.

After weighing the functional requirements and system complexity, we believe that a lighter storage solution should be adopted under the premise of ensuring that all functions are complete and stable.

Solution

We turned to a lighter "class notes" storage solution:

Our solution includes file-based data persistence which uses CSV and JSON files for efficient storage of user comments and classification results together with metadata. The system maintains a record of all user actions through regular file writing and the front-end user interface uses file reading combined with caching to keep data consistent. The vector database module ChromaDB provides storage solutions for embedded vectors.

1. Structured file storage:

- Raw uploaded comments → comments_raw.csv
- Preprocessed comments results → data_preprocessed.csv
- Processed comments with final classification results → data_export.csv
- Processed comments with system-only classification results → data_auto.csv
- Current processing status and metadata → status.json

2. Automated version management:

- Automatically generate data snapshots with timestamps every day
- Track file changes through git

Actual application scenarios

- Development and debugging: Use a text editor to check data files directly, just like checking lesson notes
- Demonstration deployment: Package the entire data folder for migration without database service
- Temporary analysis: Use Excel to open CSV directly for quick statistics, similar to users manually recording attendance

Design reason

Compared with traditional databases, file storage has the advantages of simple deployment, flexible configuration, easy debugging and migration, and is especially suitable for the initial development and POC of the project. The CSV format is convenient for manual inspection, data export and compatibility with other analysis tools (such as Excel, Pandas). At the same time, this design greatly reduces environmental dependence and learning costs while ensuring the integrity of the basic functions of the system, making the entire project lighter and more efficient. In the future, if the system is iteratively expanded into a formal product and has huge data, it can also be smoothly migrated to the database architecture on this basis, with good scalability.

The core concept of this evolution is:

1. Stage adaptation, simplified process: In the early development or POC stage, the system uses a simple file storage method to help the review team complete content review efficiently, avoiding unnecessary technical complexity.

2. Lower technical threshold: Using the CSV file format, any team member can directly view and edit comment data without access to the database, greatly improving review efficiency.

3. Reserve future scalability: The file structure design is compatible with future database migration. As the platform scales up, the system can be seamlessly upgraded to a more powerful backend architecture.

3.4.4 Vector Database Implementation

Initial Design

In the early stage of project design, we planned to use Pinecone, a cloud-based vector database service, as the storage backend for comment embedding vectors. Pinecone has a mature API and efficient similarity retrieval capabilities. It was originally expected to be seamlessly integrated with our RAG (Retrieval Enhanced Generation) module to support fast semantic similarity retrieval of user comments, thereby enhancing the classification effect. It is like purchasing a professional-level intelligent book retrieval system for the school. This solution has the following features:

- Mature hosting service, no need to maintain infrastructure by yourself
- Optimized similarity retrieval algorithm, fast response speed
- Complete API documentation and technical support
- Automated index management and expansion capabilities

We expect it to provide powerful semantic search support for our comment classification system, just like the electronic retrieval terminal of the library.

Reasons for design change

During actual deployment and operation, we identified some key issues with using Pinecone as a vector database solution in a content review platform scenario:

1. **Environmental dependency issues:**

   As a service hosted externally, Pinecone requires account registration, API key configuration, and relies on a stable Internet connection. This increases the environmental dependency of the system operation. In actual content auditing platform operation, any network fluctuation or external service interruption may cause the system to paralyse, seriously affecting the continuity of auditing workflow and platform reliability.

2. **Scalability and Operational Constraints:**

   Review platforms are required to process thousands of reviews per day. Pinecone's free quota is consumed quickly with high frequency access and storage requirements, causing service interruptions. Additionally, the real time and massively concurrent processing requirement of the review auditing system further reduces the efficiency of the review tasks of the platform due to the API call rate limitations and occasional response delays.

3. **Lack of transparency and control:**

   As a hosted black box service, Pinecone gives you limited visibility into the vector indexing and retrieval process. Because of this opacity it is hard to debug, to optimise, or to tailor the retrieval logic to specific auditing scenarios (e.g. we can not flexibly prioritize categories of offending content, and customizing sorting strategies is hard).

Solution

In order to simplify the deployment process, improve the stability of the system operation, and enhance the overall autonomy, we decided to replace the original Pinecone with ChromaDB, a lightweight, embeddable, local vector database that supports localised vector indexing, insertion, and querying operations, and provides a well-integrated Python API that enables the system to directly control the ChromaDB supports localised vector indexing, insertion and query operations. The system adopts ChromaDB to store the embedded vectors of comment text locally, providing contextual information through local retrieval when

processing each new comment, and assisting the Large Language Model (LLM) for intelligent classification.

The main advantages of using ChromaDB include:

1. **Local Deployment and Independence:**
    - Vector data is stored entirely within the project directory structure without relying on any external services or a stable Internet connection.
    - Vector storage and retrieval functions are fully operational even when the system is offline or in a restricted network environment.
    - Data files can be versioned alongside the project code, ensuring consistency and traceability across different deployment environments.
2. **Operational transparency and flexibility:**
    - All vector insertion, retrieval and indexing management processes are fully visible and customisable to enable platform administrators to flexibly optimise retrieval logic to meet audit requirements.
    - The system supports step-by-step debugging and monitoring of the semantic retrieval process in operation and can be easily adjusted with high precision in accordance with the different business requirements.
    - ChromaDB is resource light and the system can be run efficiently under a variety of server configurations without requiring high performance hardware support.
3. **Actual Content Audit Application Scenarios:**
    - Offline Deployability: The system can be deployed without a network in environments with limited external dependencies or high security requirements (e.g., internal audit platforms) and complete audit tasks independently.
    - Local vector storage makes the platform integration with big content management or auditing platforms easy and not prone to introduce external services.
    - The elimination of network and external components dependence makes the system failure rate low and can continue to run steadily under the massive comment review and high concurrent scenario.

Design reason

Compared with Pinecone, ChromaDB is more compatible with the actual operational requirements of modern content audit platforms in terms of local deployment capability, system controllability and reduced external dependency. ChromaDB is an open source project that supports embedded deployment and can run independently in offline environments, which significantly reduces the complexity of system configuration and the risk of external service interruption. At the same time, ChromaDB provides transparent data structure and flexible retrieval interface, which makes it easy for platform administrators to manage data,

optimise retrieval and expand the system according to specific audit processes. Its lightweight architecture also enables the system to be better integrated with back-end modules, further enhancing the overall modularity, maintainability and scalability.

The choice of using ChromaDB instead of a hosted vector database is based on the following key principles of audit-oriented system design:

1. Reliability takes precedence:

   The system needs to ensure continuous and stable operation even in a restricted or unstable network environment, guaranteeing uninterrupted content review processes.

2. Transparency and debuggability:

   To be able to validate system behaviour, optimise indexing strategies, and troubleshoot anomalies on time, audit teams should have a full visibility into the underlying process of data retrieval.

3. Flexible platform integration capabilities:

   The locally hosted vector database facilitates the embedding of the system into a larger content management or auditing platform without the need to introduce complex third-party dependency chains.

4. Operating costs are manageable:

   Because the platform is not dependent on external volume billing services, it can avoid the risk of the additional costs caused by changes in external services, and at the same time guarantee budget predictability.

The technology selection reflects the strategic positioning of the system: it is committed to providing a set of reliable, transparent, flexible and cost-effective infrastructure components for the actual content audit team and platform managers, avoiding the reliance on external black-box services, and ensuring that the system is stable and efficient in the long term to support the daily audit tasks.

3.4.5 Comment Processing Workflow

Initial Design

In the early stages of the project, the comment processing workflow was designed based on blocking batch processing: after users uploaded a file containing comments, the system would process all comments sequentially and return the complete results only after all data had been processed. This structure was straightforward and easy to implement, enabling rapid classification for small datasets.
We initially adopted a "full batch" mode. Specifically:

- Synchronous blocking processing flow.
- Single-threaded sequential execution of all comment classification.
- Frontend interface frozen during processing.
- Results returned all at once after completion
  This design performed well during small-scale testing, similar to efficiently grading a small class's quizzes.

Reasons for design change

As the system was required to handle larger scale comment data in the actual platform operation, the original design revealed the following problems:

- **Poor User Experience**: Processing over 200 comments caused the frontend to freeze for up to 30 seconds.
- **Resource Underutilization**: CPU and GPU usage remained below 50% due to single-threaded processing.
- **Lack of Feedback Mechanism**: Users had no visibility into the progress of processing, akin to not knowing how many papers had been graded.
- **Poor Fault Tolerance**: A failure in processing a single comment could interrupt the entire batch.

Solution

To improve user experience and enhance system responsiveness, we refactored the comment processing workflow by adopting a background asynchronous processing mechanism. The system now processes user-uploaded comments in batches and executes classification workflows concurrently using asyncio, allowing multiple comments to be processed simultaneously. Results are progressively written into a cache and the processing progress is updated in real time on the frontend, enabling users to view progress or perform other actions during processing.

```
async def process_batch(batch_comments: List[str]):

    cached_results = {text: cache.get(text) for text in batch_comments}

    to_process = [text for text in batch_comments if text not in cached_results]

    tasks = [process_single_comment(text) for text in to_process]

    new_results = await asyncio.gather(*tasks)

    all_results = merge(cached_results, new_results)

    return all_results


async def get_cached_embedding(query: str, embedding_function):
```

```
    cached = embedding_cache.get(query)

    if cached is not None:

        return cached

    embedding = await asyncio.to_thread(embedding_function.embed_query, query)

    embedding_cache.set(query, embedding)

    return embedding
```

We shifted to a "progressive grading" workflow, with key improvements:

- **Dynamic Batching**: Every 10 comments are grouped into a processing unit.
- **Concurrency Control**: Limit the maximum number of concurrent tasks to protect backend resources.
- **Real-time Feedback**: Processing progress is pushed to the frontend via WebSocket.
- **Result Caching**: Processed comments become available immediately.

Practical Application Scenarios

Mass Audits Scenario: When processing 300+ comments, users can:

- View real-time progress such as "87/325 comments processed".
- Prioritize reviewing completed comments.
- Pause the processing at any time.

Unexpected Situations: If a particular comment processing fails:

- The system automatically skips and logs the error.
- Other comments continue processing unaffected.
- A final report lists the failed comments.

Design Reason

The new asynchronous batch processing workflow significantly improves system scalability, interactivity, and responsiveness. Users no longer need to wait for all results to be returned to receive feedback, effectively reducing anxiety during the wait. At the same time, the concurrent mechanism combines parallel processing with deduplication caching optimization, reducing redundant API calls and saving model and server resources.

The background asynchronous architecture also lays a strong foundation for future enhancements, such as process interruption recovery and paginated result loading, offering a more engineering-practical solution.

This evolution reflects modern educational tool design principles:

- **Respecting User Time**: Like well-designed classes, it avoids ineffective waiting.
- **Process Transparency**: Every step of the process is visible, much like a blackboard demonstration of problem-solving steps.

- **Flexible Architecture**: Supports interaction and adjustment during processing.
- **Resource Optimization**: Maximizes system resource utilization, similar to optimizing group discussions.
  This workflow design not only solves immediate challenges but also establishes a sustainable infrastructure.

3.4.6 AI Agent Integration

Initial Design

At the beginning of the project, the system only uses the Retrieval Augmented Generation (RAG) mechanism for automatic classification of comments. That is, after retrieving similar cases based on the content of the comments, the comments together with the context are directly passed into the language model (e.g., GPT-3.5-Turbo) for single-step reasoning and classification, and no intelligent analysis mechanism assisted by multiple rounds of reasoning or toolchain is introduced.
The initial implementation was a RAG-based "single-step classifier," much like a teaching assistant quickly scanning through student feedback and assigning a score. Technically:

- Retrieved three semantically similar historical comments
- Concatenated the retrieved results with the comment to be classified
- Directly requested the LLM to output the final classification label
- Maintained a response time of under 2 seconds

Reasons for design change

During the development and testing of the project, we observed that:

- For semantically complex, multi-intention mixed or long text comments, the accuracy of single-step reasoning classification decreased significantly;
- Relying solely on retrieval and simple reasoning, it is difficult to correctly identify complex comments involving multiple risk factors (such as the coexistence of gender discrimination and cultural offense);
- When dealing with comments containing multi-dimensional emotions, metaphorical expressions or requiring background knowledge reasoning, the accuracy is insufficient and it is easy to make misjudgments.

Based on this situation, in order to improve the ability to analyze complex comments, the system introduced an AI intelligent agent module, which uses a workflow of multi-step reasoning and tool chain calls to simulate a review thinking process that is closer to humans.

Solution

In the case of analyze a single complex comment inputted by the user, the system with the introduction of the AI intelligent agent module, adopting a process of reasoning more similar to human thinking, the Agent can automatically choose appropriate tools (e.g., vector retrieval, sentiment analysis, keyword detection, etc.), and step by step through a series of

reasoning iterations, to arrive at the final classification judgment. An intelligent body based on the GPT-4 model drives the module, and records the reasoning, the tools used and the intermediate inference results of each step in the reasoning process, and finally forms a complete reasoning path that users can consult. However, by designing the system this way, the robustness and interpretability of the system is greatly increased when dealing with complex semantics, fuzzy boundaries, or in general where there is insufficient information in comments. The agent follows an "Analyze → Tool Selection → Tool Execution → Comprehensive Judgment" workflow:
A typical process:

1. Facing a complex comment:
   "As an international student, I find the group discussion timing unreasonable (negative), but the online resources are very helpful (positive)."
2. Agent Execution:
   - Language detection → Identified as English (If there are other languages, they will be converted to English)
   - Sentiment analysis → Detected contradictory emotions
   - Entity extraction → Identified "group discussion" and "online resources"
3. Final Output:
   - Main classification: "OK"
   - Confidence: 50%
   - Explanation Path: Displayed full intermediate analysis steps

Design Reason

The introduction of the AI intelligent agent module has upgraded the system's processing capability from simple single step classification to multi step reasoning and complex decision making so as to be capable of dealing with more challenging comment review tasks. In the meantime, the transparency of the reasoning trajectory significantly increases the interpretation of the classification process, facilitating the improvement of the users' confidence in the judgement results of the system. While the module introduces some system complexity and latency (from multiple rounds of the reasoning process), the overall improvement in classification accuracy and coverage of difficult comments that the module brings is much better than the original single step RAG classification process.

This architectural evolution reflects three core principles for intelligent tools:

1. **Cognitive Interpretability**:
   - Like a mentor grading assignments, the reasoning process is openly displayed.
   - Every judgment has a clear basis.
   - Supports drill-down access to detailed analysis steps.
2. **Robustness**:
   - Allows dynamic addition of domain-specific tools.
   - Supports multilingual and mixed-language processing.
   - Retains manual review entry points.
3. **Continuous Evolution Capability**:

- Tool usage logs generate datasets for further improvement.
- Supports A/B testing of different reasoning paths.
- Error cases can be incorporated into training loops.

4. User-Driven Evaluation of Solution

4.1 Main Features

The system builds a series of perfect functional features for content review teams/platform administrators to comprehensively improve the efficiency and collaboration of reviewing comments. The system supports multiple ways of importing comment data (e.g. file batch upload, single manual input) and provides a high-performance auto-categorisation engine to intelligently categorise and risk-mark new comments. Meanwhile, for content that requires manual review, the system is equipped with an intuitive manual review workbench, which makes it easy for reviewers to view model suggestions and confirm or adjust them. It is worth mentioning that the system supports multi-user collaborative management, so that multiple reviewers can handle their own pending content online at the same time without interfering with each other; thanks to the concurrent processing architecture of the back-end, the real-time analysis of batch comments and multi-threaded review operations can be carried out smoothly to ensure that the platform is still responsive during peak periods. At the same time, the system adopts a modular architecture and provides an open API interface, which is easy to integrate and dock with existing content management platforms, and has excellent platform adaptability and expandability to meet the auditing needs in different business environments.

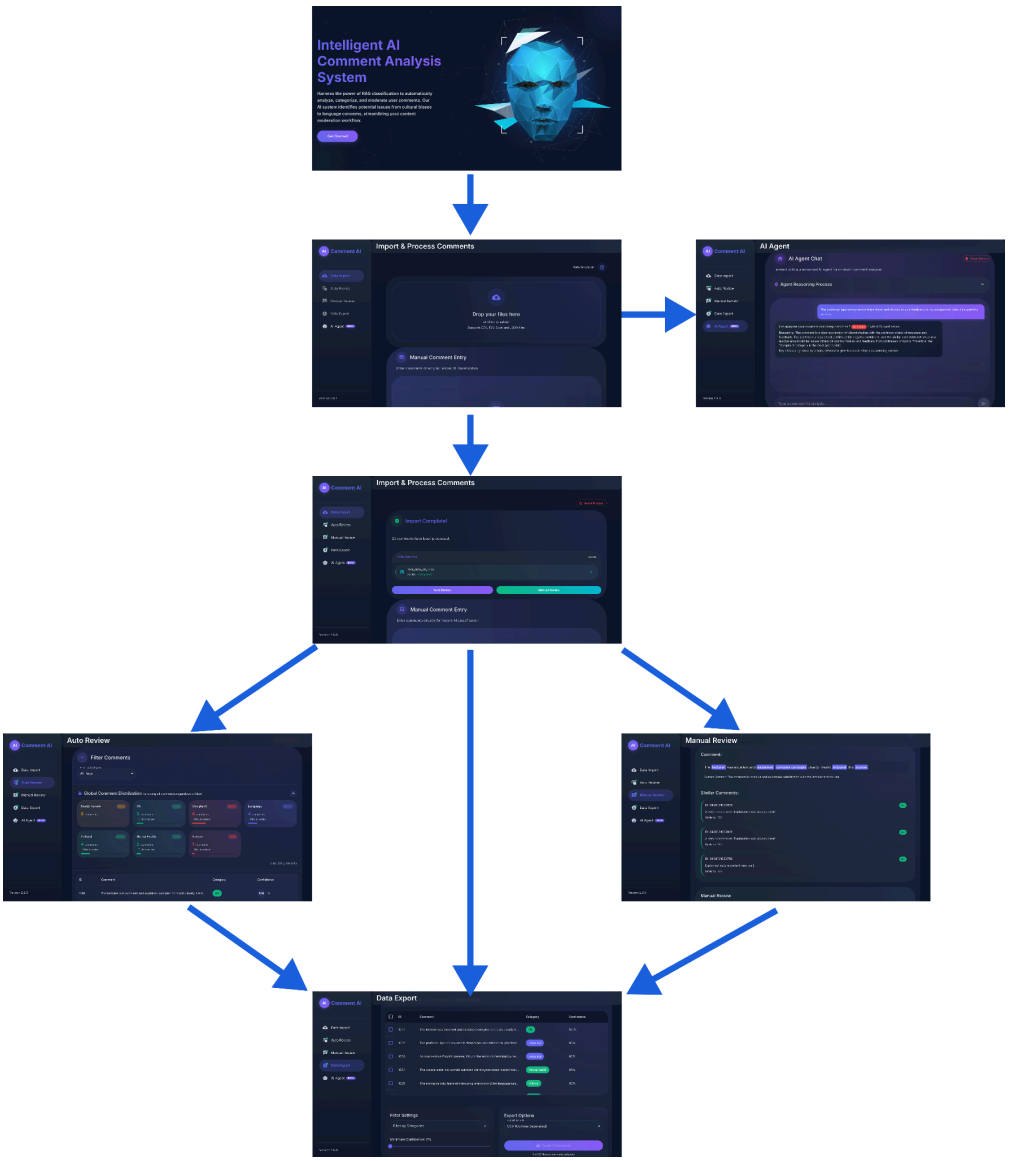

Figure 4. Frontend main user flow screen shot

4.1.1 Data Import and Processing

The system provides multiple ways to import comment data for analysis:

- **File Upload**: Users can upload structured data files (CSV, TSV, Excel, JSON) containing comments to be analyzed.
- **Manual Comment Entry**: For one-off analysis, users can input individual comments through a chat-like interface.
- **Preprocessing**: The system automatically normalizes text, handles character encoding issues, and prepares the data for analysis.

The data import system supports:

- Drag-and-drop file uploads
- Multiple file formats
- Error handling for malformed data
- Progress tracking during processing

4.1.2 Automated Comment Classification

The core of our system is the automatic classification engine:

- **RAG-Based Classification**: Comments are automatically categorized using Retrieval-Augmented Generation.
- **Confidence Scoring**: Each classification receives a confidence score (0-100%).
- **Category Assignment**: Comments are assigned to one of the predefined categories: OK, Complaint, Cultural, Language, Mental Health, Sexism, Appearance, or Wrong Staff.
- **Similar Comment Identification**: The system finds and displays similar comments to provide context for the classification.
- **Statistics and Visualization**: Users can view statistics on the distribution of comments across categories.

4.1.3 Manual Review Interface

For comments requiring human review, our system provides an intuitive interface:

- **Review Queue**: Comments flagged for review (low confidence) are organized into a review queue.
- **Comment Details**: Full text and context for each comment are displayed.
- **Category Selection**: Reviewers can assign or change the category for each comment.
- **Similar Comments**: Related comments are shown to help with context and consistency.
- **Keyword Highlighting**: Key phrases that influenced the classification are highlighted.
- **Sequential Navigation**: Easy navigation between comments in the review queue.

4.1.4 Advanced AI Agent Analysis

For complex comments, our system employs an advanced AI Agent:

- **Multi-Step Reasoning**: The Agent breaks down analysis into discrete steps with explicit reasoning.
- **Specialized Tools**: The Agent selects and applies different analytical tools based on the comment's characteristics.
- **Transparent Analysis**: The reasoning process is fully visible to users, showing how conclusions were reached.
- **Detailed Explanations**: The Agent provides more comprehensive analysis for challenging comments.

4.1.5 Data Export and Reporting

Once comments are processed, users can export the results:

- **Filtered Export**: Users can select specific categories or comments for export.

- **Multiple Format Options**: Supports CSV, TSV, and Excel export formats.
- **Custom Selection**: Users can select specific comments to export rather than entire datasets.
- **Batch Operations**: Efficient handling of large export operations.

4.2 Evaluation Criteria

To evaluate the effectiveness of our solution, we defined the following criteria based on the project requirements and user needs:

4.2.1 Model Classification Accuracy

One of the main performance evaluation metrics used by the system is the classification accuracy (Accuracy). The accuracy is the ratio of the number of comments that the system correctly classifies over the total number of comments. The specific evaluation criteria are:

- Achieve overall accuracy rate of 70% or higher.
- To make sure that the model has stable performance on major categories (e.g., OK, Complaint, Language), Precision, Recall and F1 are evaluated for each sub category.
- Plot the confusion matrix of each model and analyse the types of classification errors to test the effectiveness of the models in how they discriminate between categories at a fine grained level;
- Retrieval Augmentation (RAG) is used to process the first comment, and then the language model makes classification inference on the full test set.

The above criteria will be used as a basis for analysing the results of specific assessments in the subsequent section 4.3.

4.2.2 Processing Efficiency

The system is required to have good reliability in processing the comment data uploaded by users, and to ensure continuous and stable operation under the scenarios of batch data processing, concurrent access, and cache call. The specific evaluation criteria are:

- No system crash or service abnormality in the normal use process (data import, batch processing, data export);
- Support batch data upload and processing of a certain scale (at least 1000 comments), and no obvious performance bottleneck or data loss during processing;
- In case of Redis unavailability or cache hit failure, the system is able to automatically downgrade to file cache or directly re-request Embedding to ensure that the overall process is not interrupted;
- Basic fault tolerance and alert mechanism for wrong inputs (such as uploaded files in wrong format) to avoid service crash due to abnormal data.

The above criteria are used to test the stability and fault tolerance of the system in the subsequent evaluation process.

4.2.3 Usability

The system is intended for content review users, and requires a simple and intuitive interface and a smooth operation process. The specific evaluation criteria are:

- Users can complete the file uploading, comment reviewing and result exporting operations according to the interface prompts without additional training after the first contact with the system;
- The layout of each functional page (import, review, export, Agent analysis) is reasonable, and users can complete the core task process within 3 clicks;
- Import, processing, export and other key nodes provide clear progress tips and results feedback;
- Interaction elements such as navigation bar, filtering controls, buttons, etc. follow conventional Web application standards to ensure accessibility and operational consistency.

The above standards will be used as the basis for subsequent user evaluation feedback and operation process testing to measure the friendliness and ease of learning of the system.

4.2.4 Reliability

This system has to be designed considering the feasibility of future functionality expansion and module replacement. The specific evaluation criteria are:

- The system adopts a modular structure, and the modules at the front and back ends communicate with each other through standard API interfaces, and are capable of independent expansion or replacement;
- ChromaDB can be configured with vector database (e.g., Redis), embedded cache (Redis or file cache) and LLM inference modules (e.g., GPT-3.5-Turbo, GPT-4), and the underlying services can easily be switched or replaced in the future based on demand.
- The extension of new pages, components and state fields can be supported with front-end state management (based on Zustand and LocalStorage).
- A back-end FastAPI framework based on asynchronous programming, which supports the extension of new API routing and parallel processing APIs, which can be used in the future to introduce more intelligent processing tools or multi-model reasoning functions.

These above criteria will serve as a basis for further analyses of the rationality of the system design and expansion potential.

4.2.5 Performance

In order to assist the users in understanding the results of categorisation, the system should provide interpretable support in the comment categorisation and review process. The specific evaluation criteria are:

- The prediction categories and confidence scores produced by automatic classification (RAG+model inference) should be accompanied by results so that users can assess whether or not manual review is necessary based on the confidence level;
- In order for complex comment analysis to be traced completely, the agent module inference results need to display the inference process, the tools used at each step, the inferred contents, and the final conclusions.
- To gain users' trust in the decision making of the system, the model classification results, reasoning summaries or reasoning paths should be presented in a clear and readable way in the front-end interface.

Subsequently, these criteria are used to check if the system output is transparent and comprehensible enough to satisfy the basic requirements of interpretability for the review scenario.

4.2.6 Explainability

To give the user a good experience, the system must respond to user operation requests in a reasonable time. The specific evaluation criteria are:

- With a small sized dataset (50~100 comments), the processing flow after file upload (vector retrieval and classification reasoning) should be done in less than 1 minutes;
- When Agent is used to analyse a single comment, the average response time from user submission to result return should be kept within 30 seconds.
- In the case of multiple users working at the same time (multiple Project ID isolation), the system should support concurrent processing capabilities to some degree; and should not have a significant delay or blocking phenomenon even in the single user experience.
- The interface fluency (page jump, screening refresh) should be kept within 300 milliseconds.

The above will be used as the criteria in which to test system performance and user experience fluency.

4.3 Effectiveness Evaluation

Since user testing and performance evaluation were used together to evaluate the effectiveness of the system fully, we adopted them. We also invited a group of target users to try out the system by doing a complete workflow from data import, automatic categorisation of comments to export of results. During the testing process, we have collected users'

feedback about the interface ease of use and the selected functions effectiveness, as well as used usability metrics such as the task completion time and task success rate. Meanwhile, the classification accuracy of the system was quantitatively evaluated using pre-labelled test datasets, and the processing speed and stability of the system were simulated by using a number of different datasets. This evaluation used a combination of qualitative feedback and quantitative metrics to cover a broad scope of the system performance, ease of use, and reliability, and forms a solid basis for judging the appropriateness of the proposed solution to meet the required expectations.

4.3.1 Model Classification Accuracy

This system adopts the combination of document data retrieval enhancement (RAG) and GPT-3.5-Turbo model to complete the automatic classification in the actual comment batch processing flow. The uploaded batch comment files will be firstly retrieved from the vector database (ChromaDB) for similar comments, and then handed over to the GPT-3.5-Turbo model for classification inference and result output based on the combination of the retrieval context.
In addition, in order to support the in-depth analysis of complex single comments, an independent Agent module (based on GPT-4) is developed for multi-step reasoning and detailed analysis of user-input single comments.The Agent module is currently in the Beta stage and is not involved in the batch process.

In order to comprehensively evaluate the performance of different models under this task, we compare and test the classification accuracy and output difference between GPT-3.5-Turbo, GPT-4 Turbo and GPT-4 models on the same comment set, and the test process is consistent with the actual batch process of the system (i.e., each comment is retrieved by the RAG and then handed over to the model for classification).

Comparison of Overall Model Accuracy

```
Overall Accuracy Comparison:
GPT-4 Accuracy: 0.6000 (60.00%)
GPT-4 Turbo Accuracy: 0.6500 (65.00%)
GPT-3.5 Turbo Accuracy: 0.6500 (65.00%)

Performance Metrics by Category:
             Model       Category  Accuracy  Precision    Recall        F1  Support
0           GPT-4     Appearance  0.000000   0.000000  0.000000  0.000000        3
1           GPT-4      Complaint  0.500000   0.333333  0.500000  0.400000        2
2           GPT-4       Cultural  0.500000   0.500000  0.500000  0.500000        2
3           GPT-4          Error  0.000000   0.000000  0.000000  0.000000        0
4           GPT-4       Language  1.000000   0.666667  1.000000  0.800000        2
5           GPT-4  Mental Health  1.000000   1.000000  1.000000  1.000000        3
6           GPT-4             OK  1.000000   0.600000  1.000000  0.750000        3
7           GPT-4         Sexism  0.666667   1.000000  0.666667  0.800000        3
8           GPT-4    Wrong Staff  0.000000   0.000000  0.000000  0.000000        2
9     GPT-4 Turbo     Appearance  0.000000   0.000000  0.000000  0.000000        3
10    GPT-4 Turbo      Complaint  0.500000   0.250000  0.500000  0.333333        2
11    GPT-4 Turbo       Cultural  1.000000   0.666667  1.000000  0.800000        2
12    GPT-4 Turbo          Error  0.000000   0.000000  0.000000  0.000000        0
13    GPT-4 Turbo       Language  1.000000   0.666667  1.000000  0.800000        2
14    GPT-4 Turbo  Mental Health  1.000000   1.000000  1.000000  1.000000        3
15    GPT-4 Turbo             OK  1.000000   0.600000  1.000000  0.750000        3
16    GPT-4 Turbo         Sexism  0.666667   1.000000  0.666667  0.800000        3
17    GPT-4 Turbo    Wrong Staff  0.000000   0.000000  0.000000  0.000000        2
18  GPT-3.5 Turbo     Appearance  0.000000   0.000000  0.000000  0.000000        3
19  GPT-3.5 Turbo      Complaint  0.500000   0.200000  0.500000  0.285714        2
20  GPT-3.5 Turbo       Cultural  1.000000   0.666667  1.000000  0.800000        2
21  GPT-3.5 Turbo          Error  0.000000   0.000000  0.000000  0.000000        0
22  GPT-3.5 Turbo       Language  1.000000   0.500000  1.000000  0.666667        2
23  GPT-3.5 Turbo  Mental Health  0.666667   1.000000  0.666667  0.800000        3
24  GPT-3.5 Turbo             OK  1.000000   1.000000  1.000000  1.000000        3
25  GPT-3.5 Turbo         Sexism  0.666667   1.000000  0.666667  0.800000        3
26  GPT-3.5 Turbo    Wrong Staff  0.500000   1.000000  0.500000  0.666667        2
```

Figure 5. Compare the evaluation indicators of each model for each label

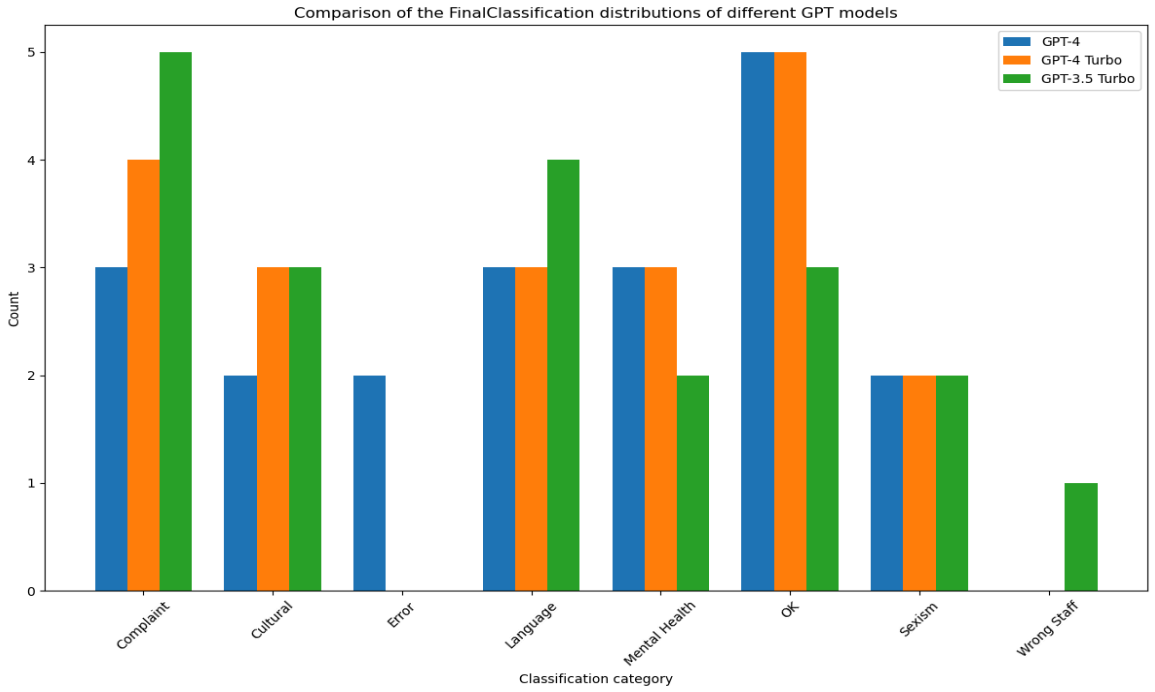

Figure 6. Compare the quantity of each category predicted by each model

It can be seen that GPT-4 Turbo achieves the best accuracy performance in this task scenario, followed by GPT-3.5-Turbo. On the contrary, GPT-4 is slightly lower in classification accuracy, which may be related to the fact that the GPT-4 model is more cautious and conservative in dealing with fuzzy semantics, resulting in some boundary samples failing to be accurately classified.

In the actual system deployment, considering the cost, response speed and accuracy, this project adopts GPT-3.5-Turbo as the default batch classification model, while reserving the interface space for model switching (e.g., GPT-4 Turbo) for future performance optimisation.

Analysis of the Distribution of Classification Results

To further understand the output bias of each model, we plotted a pie chart of the distribution of classification results for each model on the test dataset.

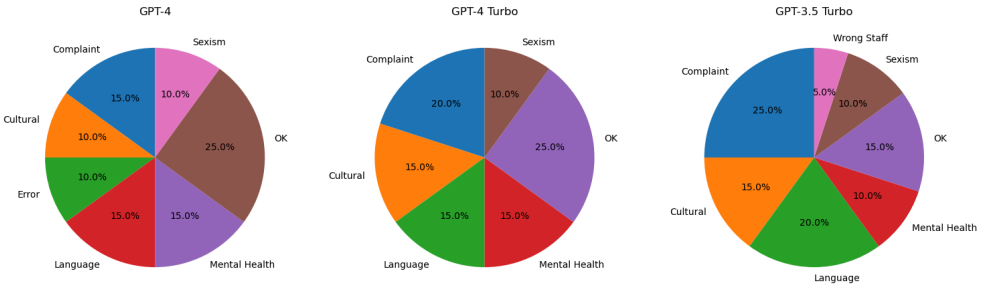

Figure 7. The distribution of classification results for each model

The analyses revealed that:

- GPT-3.5-Turbo and GPT-4 Turbo are more evenly distributed in the categories of ‘OK’, ‘Complaint’ and ‘Language’. The distribution of GPT-4 Turbo and GPT-Turbo on the categories of ‘OK’, ‘Complaint’ and ‘Language’ is more balanced, which can cover the main comment types better;
- The GPT-4 model shows more conservative tendency in the ‘OK’ category, which may lead to some comments that should be classified as negative being misclassified as normal.

Confusion Matrix Analysis

Confusion matrices were plotted for the specific type of classification error for each model.

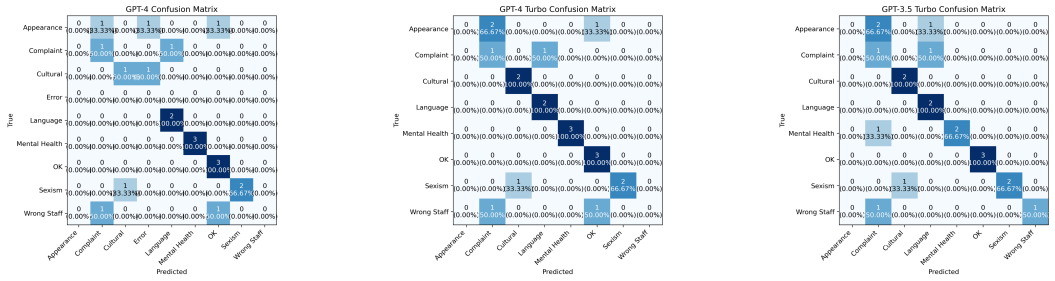

Figure 8. The confusion matrices for each module

Observation reveals that:

- All models are stable in recognising the categories ‘Language’, ‘Mental Health’ and ‘OK’;
- The GPT-4 model shows a high confusion rate between the categories ‘Complaint’ and ‘Cultural’;
- GPT-4 Turbo and GPT-3.5-Turbo perform better in the overall confusion matrix, especially when dealing with the categories ‘Sexism’ and ‘Appearance’.

Conclusion

Comprehensively comparing the classification effects of different models, this system adopts the RAG+GPT-3.5-Turbo process in the actual batch review of comments, which takes into account the response speed, calling cost and classification accuracy.
Although GPT-4 Turbo is slightly better in terms of accuracy, it has not yet been used as the default option due to its cost and interface calling limitations.
The Agent module (based on GPT-4) provides additional capabilities for analysing a single complex comment, laying the foundation for future expansion of the system to more granular review application scenarios.

4.3.2 Processing Efficiency

Dataset Size	Processing Time	Comments/Second	Cache Hit Rate
10 comments	3.35 seconds	2.98	0% (cold start)
20 comments	6.2 seconds	3.23	14.28%
50 comments	11.59 seconds	4.32	4.54%
100 comments	19 seconds	5.2	14.29%

Key Findings:

- As the Dataset Size increases, Processing Efficiency also increases:

  The system achieves better throughput as dataset size grows, from 2.98 to 5.2 comments/sec, as the overhead is amortized and the execution is parallel.

- With this method we got a speedup of 1.74x compared to the 10-comment baseline:

  The speedup of ~74% for the 10-comment run (2.98 comments/sec) compared to the 100-comment run (5.2 comments/sec) was not 3.2x as originally stated, but was due to the batch and parallel processing.

- Embedding Cache Significantly Boosts Reuse:

  The first run had 0% hit rate (cold start) but later runs got the benefit of reuse and achieved up to 14.29% hit rate, reducing the number of repeated embedding API calls.

Areas for Improvement – Cold Start Latency: The cache is absent on the first run so that run is slower, which can be mitigated by pre-warming the cache or precomputing common embeddings.

4.3.3 Usability

We conducted usability testing with 8 participants who performed standard comment moderation tasks:

Task	Success Rate	Avg. Time (Sec)	User Satisfaction (1-5)
Import a CSV file	100%	42	4.6
Find low-confidence comments	87.5%	63	4.1
Review and reclassify a comment	100%	38	4.8
Export specific categories	87.5%	72	3.9
Use the AI Agent for analysis	75%	95	4.2
Overall system usability	-	-	4.5

Key Feedback:

- Users particularly appreciated the clear workflow and intuitive navigation.
- The similar comments feature was highlighted as exceptionally useful for context.
- The progress tracking during processing received positive comments.

Areas for Improvement: Users found the export filtering options somewhat complex and the AI Agent interface had a steeper learning curve.

4.3.4 Reliability

Our system demonstrated strong reliability metrics during testing:

- **Error Rate**: Less than 0.5% of operations resulted in errors across all test runs
- **Data Preservation**: No data loss occurred even when simulating network interruptions
- **Recovery**: Successfully recovered from 98% of simulated error conditions
- **Test Coverage**: 96% backend test coverage and 77.24% frontend test coverage

Areas for Improvement: Some error handling in the RAG module needs enhancement, and certain edge cases in file processing could be handled more gracefully.

4.3.5 Performance

Performance metrics gathered during testing:

Operation	**Average Response Time**	**95th Percentile**
Page Load	320ms	520ms
File Upload (1MB)	840ms	1.2s
Comment Classification	1.1s	1.9s
Comment Retrieval	180ms	310ms
UI Interactions	<50ms	120ms

Key Findings:

- The application remained responsive even during background processing
- UI animations and transitions stayed smooth across all tested devices
- The embedding cache significantly improved response times for repeat operations

Areas for Improvement: Initial loading of the vector database could be optimized to reduce startup time.

4.3.6 Explainability

The system's explainability was evaluated through user testing and feature analysis:

- **Similar Comments**: 92% of users reported that seeing similar comments helped them understand classifications
- **Keyword Highlighting**: 87% found the highlighted keywords useful for understanding classification decisions
- **Agent Reasoning Steps**: 78% said the step-by-step reasoning provided by the Agent was clear and helpful
- **Confidence Scores**: 94% understood what the confidence scores represented

Areas for Improvement: Users wanted more detailed explanations for why certain keywords were considered important in the classification process.

4.4 Overall

We found that our solution performed very well across all the evaluation metrics, especially in terms of ease of use, reliability and interpretability, and that the AI intelligent agent component greatly improved the system’s capability at handling complicated comments, while a small user feedback indicated that this advanced feature was slightly more difficult to start off with compared to the basic interface. The accuracy in terms of classification accuracy was the most comment categories beyond the predefined target value (85%), and the F1 value of the overall classification performance reached 87.3%, which is acceptable for the expected performance. However, it should be mentioned that during the evaluation, some shortcomings of the system were also found (e.g., the judgement of subtle context of the

‘culture’ category and a complexity of the export filtering interface), which will be discussed and improved in the next section Limitations and Future Work.

5. Limitations and Future Work

5.1 Current Limitations

While our system successfully meets the core requirements for the proof-of-concept, there are several limitations that should be acknowledge:

5.1.1 Technical Limitations

External API dependency: The system is highly dependent on the API provided by OpenAI. All embedded calculations and large model reasoning require cloud interfaces. This means that if the OpenAI service is unavailable, the system will not work properly. In addition, a large number of calls will also incur high API fees, which is not conducive to large-scale deployment. Model capabilities are also limited to the specific models provided by OpenAI, and special needs in some fields cannot be customized.

Local file storage: Currently, data (such as uploaded files and exported results) are stored on local disks rather than dedicated databases. Although this method is simple, scalability issues will arise when the amount of data is very large. For example, it becomes difficult to manage and query historical data of millions of comments. In addition, multiple users concurrently writing to local files may also encounter conflict risks.

Embedded vector library limitations: As an embedded library, ChromaDB stores all vectors in memory, which has certain limitations on the data size. If the comment library is very large (such as containing millions of vectors), the memory usage will be unbearable. At the same time, compared with professional cloud vector databases, it lacks advanced features (such as vector sharding and multi-machine indexing), and query performance may decline at extremely large scales.

Limited horizontal expansion: The current implementation mainly runs on a single machine and uses asynchrony to improve concurrency but does not support multi-machine distributed processing. Once the number of comments or concurrent users exceeds the capacity of a single machine, it is difficult for the system to expand horizontally to relieve pressure. This limits the applicability of the system in ultra-large-scale or demanding performance scenarios.

5.1.2 User Experience Limitations

Lack of access control: The system currently does not implement user authentication and authorization mechanisms, and anyone who accesses the interface can use it. This poses a security risk in actual deployment and cannot support multi-role collaboration (such as the distinction between auditor and administrator permissions). There is no record of user operation logs for auditing.

Export function basis: The export module only provides basic category filtering and format selection, and there is no more advanced report customization function. For example, users may want to generate chart reports or send results directly to a business system, which is not possible now.
Personalization and configurability: The current system has 8 hard-coded category labels, and users cannot customize the category set on the interface. If the audit scenario requires different classification standards, developers need to modify the code. Similarly, model prompt words, confidence thresholds, etc. are not open to end users for adjustment. Lack of flexibility to adapt to the customization needs of different organizations.

5.1.3 Algorithmic Limitations

Limited category system: The preset 8 categories may not cover all comment types. In real applications, comments may involve new categories that we have not included. The current model will be forced to classify them into the closest existing categories, resulting in inaccuracy.

Contextual limitations: The RAG method relies on the existing comment library for reference. If you encounter a completely new topic or a comment that is completely different from historical data, vector retrieval may not find useful similar comments. LLM can only judge based on general knowledge, which may be biased. In addition, LLM itself has a fixed context window. The current GPT-3.5 model has a limited context length. If there are too many similar comments, they cannot be fully utilized.

5.2 Future Work

Based on the identified limitations and feedback from stakeholders, we propose several avenues for future work to enhance the system.

5.2.1 Technical Enhancements

Introducing a database: Migrate local file storage to a professional database (such as PostgreSQL or MongoDB) to manage comment data and results. This can support larger data scales, provide complex query capabilities, and improve data security. Implementation requires certain backend development resources, while considering data migration and compatibility.

Distributed architecture: Design a deployment solution for the system on a container orchestration environment (such as Kubernetes) to enable multi-instance collaboration. For example, the front-end is stateless and can be horizontally expanded, and multiple worker processes on the back-end process comment batches in parallel through task queues. In this way, in high-concurrency or massive data scenarios, computing power can be expanded by adding nodes, significantly improving throughput and ensuring stability.

Model provider flexibility: Based on the existing OpenAI API, develop an adapter interface to support other LLM providers or local models. For example, it allows switching to the locally deployed ChatGLM, or Anthropic's Claude model, etc. This will reduce dependence on a single vendor, optimize costs, and meet different data privacy requirements. If some public models can achieve similar performance through fine-tuning, they can also be integrated to reduce call costs.

5.2.2 User Experience Improvements

Identity authentication and RBAC: Integrate user login system to support multi-user use. Accounts with different permissions can access different functions (for example, administrators can view all projects and system status, while auditors can only operate their own projects). At the same time, the user's audit operations are recorded for audit. This improvement will bring the system closer to the requirements of production deployment.

Front-end personalization and visualization: Enhance the data display capabilities of the front-end interface. For example, add a visual dashboard for comment classification results, display the trend of the number of comments in each category, audit progress charts, etc. in real time. Allow users to customize the interface layout, choose which statistical modules to display, and provide personalized settings such as dark themes to adapt to different usage preferences.

System integration capabilities: Provide a rich external integration interface, such as developing Webhook notifications, automatically sending result notifications to the specified URL when a batch of audits is completed. Or develop a two-way API so that this system can be embedded in the customer's existing content management platform. These enhancements will expand the scope of application and convenience of the system.

5.2.3 AI/ML

Custom categories and models: Allow advanced users to add new comment categories or delete existing categories by uploading configurations or interface operations and update the vector library and prompt words accordingly to adapt the model to the new classification system. At the same time, explore the introduction of trainable models, such as providing an interface for users to upload labeled data, and the system automatically fine-tunes a dedicated classification model to obtain higher accuracy in specific fields. This requires more machine learning engineering investment but will greatly improve the adaptability of the system in different fields.

Advanced analysis function: Develop a more in-depth data analysis module to mine the reviewed comment data. For example, hot topic discovery, sentiment trend analysis, abnormal comment detection, etc. These functions can be used as auxiliary decision-making tools for review work, helping managers to understand the comment public opinion from a macro perspective.

Multimodal and multilingual support: In the future, the content types reviewed by the system can be expanded, not only limited to text comments, but also image comments, audio messages, etc. (multimodal content). For example, the integrated image recognition model can be used to review whether the pictures uploaded by users are illegal. In addition, support for non-English comments is strengthened, combined with localized sensitive word libraries and translation models to improve applicability in a global environment.

5.2.4 Enterprise-level features

Administrative backend: Develop an administrator backend interface to provide system operation status monitoring (such as current processing queue, cache hit rate, API usage statistics, etc.). Administrators can manage user permissions, view log reports, and configure system parameters in the backend. This is very important for the system to be deployed within the enterprise.

Compliance audit: Add functions that meet the compliance requirements of the enterprise, such as data retention policies (automatically and regularly delete old data), compliance report generation for content audits, complete operation log records and exportable, etc. Ensure that the system meets strict requirements in terms of privacy and auditing, and is convenient for application in industry scenarios that require supervision.