ChatDB Project Final Report

Group:

ChatDB 39

Group Members:

Yijian Jin

Hongyu Yu

Zilu Wang

Date of Submission:

May 9, 2025

Table of Contents

Introduction

This project aims to create a ChatDB, a database interface that allows users to interact with multiple databases using natural language queries. The system must support at least two different database instances (e.g., MySQL and MongoDB, specifically for our project) and correctly handle queries, joins, and modifications across all databases.

When complete, the system will include:

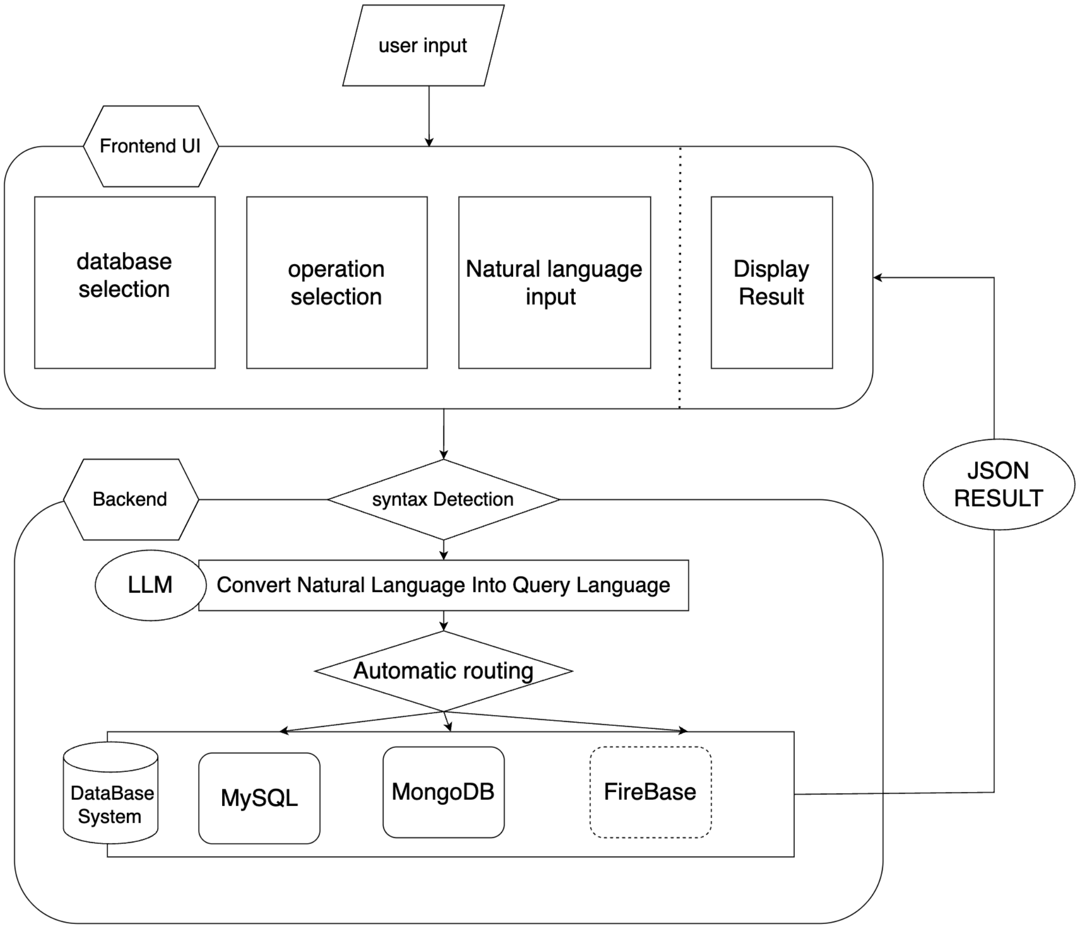
- A natural language interface (NLI) that translates user queries into executable SQL or NoSQL queries.
- Support for at least two database instances: MySQL (SQL) and MongoDB (NoSQL).
- A web-based UI to facilitate user interaction.
- A unified backend API that can use the result from NLI to support multiple database instances and handle queries, joins, and modifications across all databases.

Planned Implementation (From Project Proposal)

- Natural Language Processing and Query Translation
  - Users will input queries in natural language (English).
  - LLM converts user input into a structured SQL/NoSQL query.
  - ***Important**:* The backend routes the query to the appropriate database.
    - MySQL: relational queries
    - MongoDB: document-based queries
    - Firebase: real-time JSON-based queries
- Backend API and Query Execution
  - Using Flask or FastAPI to process user input and upload it to the NLM API for query translation.
  - Validate the returned SQL/NoSQL query before execution.
  - Route query to the corresponding database based on query type, and return the result in JSON format for the frontend showcase.
  - Implement database connectors between databases.
  - Performance optimization and error handling.
- Database Setup
  - Deploy and configure three databases of our choice:
    - MySQL (Structured Data, RDMS)
    - MongoDB (Document-based NoSQL)

        - Firebase Realtime Database (JSON-based NoSQL)
    - Sample data collection
        - Samples include users, products, orders, reviews, transitions, etc. The goal is to fulfill all three databases for testing purposes.
        - potential dataset-gathering source list
            - Kaggle/Google Cloud Public Datasets
            - Best Buy API
            - Twitter API
    - Consistency across all databases must be ensured
- Web-Based User Interface
    - The front-end will be developed using React.js
        - Key user interaction features include a text input field for user queries, windows displaying query results in a structured table format, and options to select dropdowns or filters (if applicable).
    - Frontend-Backend Communication
        - Frontend will send requests to the backend API → backend processes the query → executes query → returns result → Frontend UI renders the query result in an intuitive format.
- Testing, Validation, and Optimization
    - Verify that the NLP-generated queries are usable and match expected outputs when interacting with databases.
    - Test the speed and efficiency of queries across different databases.
    - Test frontend user interaction and front-end backend interactions.
    - security measures (if applicable and time permits)
    - Optimize LLM API calls to reduce cost and latency.
    - Performance optimization measures (asynchronous processing, batch processing)

Architecture Design



[1]

Implementations

The backend of this project is built with FastAPI and serves as a unified API layer that allows users to interact with three different databases: MySQL, MongoDB, and Firebase[2], using natural language queries, schema exploration requests, and data modification commands. When a request is received, the backend uses Google Gemini 2.0 Flash via the Genai client to convert the user's natural language input into structured queries or modification instructions tailored for each database type. It then routes these queries to the appropriate database connector modules (eg, mysql_connector.py), executes them, and merges or formats the results as needed. The backend also supports schema exploration (listing tables/collections, showing schemas, and fetching sample data) and modification operations (insert, update, delete) for all three databases, returning results in a consistent JSON format for the frontend to display.

[1] Firebase Realtime Database has been excluded from the final implementation due to an update in the project requirements specifying the use of only two databases. While the Firebase query and modification functionalities remains in the backend, they were not fully tested nor verified. Usability is therefore uncertain.

[2] Refer to footnote #1.

The frontend is built using React.js, providing functions such as natural language query, multi-database selection, query type switching, and result display. The entire interface focuses on user experience and responsive design, supporting three operations: query execution, schema exploration, and data modification.

Functionalities

- Data Integration
    - MySQL Loader (load_airbnb_mysql.py)
        - Loads Airbnb CSV data into a MySQL database, splitting the data into normalized tables: Hosts, Listings, and Reviews. It handles data cleaning, type conversion, and enforces foreign key relationships.
    - MongoDB Loader (load_airbnb_mongo.py)
        - Loads the same CSV data into MongoDB, distributing fields into three collections: listings_meta (main listing metadata), amenities (array of amenities per listing), and media (image URLs and related fields). It parses and structures data for NoSQL storage.
    - Firebase Loader (load_airbnb_firebase.py)
        - Loads the CSV data into Firebase Realtime Database, organizing it under /listings (with nested pricing and availability objects) and /hosts. It handles type conversion and normalization for Firebase's JSON tree structure.
- Database Connector Modules
    - MySQL Connector (mysql_connector.py)
        - Provides functions to connect to MySQL, run queries, validate tables, fetch schemas, and perform data modifications (insert, update, delete). It wraps errors as HTTP exceptions for FastAPI.
            - get_connection(): Establishes a connection to MySQL using the config.
            - query_mysql(sql_query): Executes a SELECT query and returns results as a list of dicts.
            - validate_table_exists(table_name): Checks if a table exists.
            - get_table_schema(table_name): Returns the schema (columns and types) of a table.
            - modify_mysql(sql_query): Executes INSERT, UPDATE, or DELETE statements, supports multiple statements.

- MongoDB Connector (mongodb_connector.py)
  - Provides functions to connect to MongoDB, run queries (including aggregation pipelines), convert BSON types to JSON, and perform modifications (insert, update, delete) on collections. Handles collection validation and error reporting.
    - get_database(): Connects to the MongoDB database.
    - get_collection(collection_name): Returns a collection object.
    - convert_objectid_to_str(doc): Recursively converts BSON ObjectIds and non-serializable types to strings for JSON compatibility.
    - query_mongodb(mongo_filter, collection_name): Executes find or aggregation queries based on the filter.
    - modify_mongodb(mod_query, collection_name): Handles insert/insertMany, update/updateMany, and delete/deleteMany operations.
- Firebase Connector (firebase_connector.py)
  - Handles Firebase initialization, node reference retrieval, querying (with filtering and ordering), and modifications (insert, update, delete) for both /listings and /hosts nodes. Ensures correct data normalization for Firebase's structure.
    - initialize_firebase(): Initializes the Firebase app with credentials.
    - get_reference(node): Returns a reference to a Firebase node.
    - query_firebase(node, query_obj): Retrieves data from a node, supports filtering and ordering.
    - modify_firebase(node, key, operation, data): Handles insert, update, and delete operations for listings and hosts.

- FastAPI Service (app.py)
  - Schema Exploration (/explore)
    - The /explore endpoint first calls the Google Gemini model to classify the user's intent (such as listing tables, showing schema, or fetching sample data). Based on the determined database type and action, it calls the appropriate connector function: for MySQL, it uses functions like get_mysql_tables(), get_table_schema(), and query_mysql(); for MongoDB, it uses get_mongodb_collections(), get_mongodb_schema(), and get_mongodb_sample(); for Firebase, it uses get_firebase_nodes(), get_firebase_schema(), and get_firebase_sample(). These connector functions

        handle the actual database queries and return the results to the endpoint, which then formats and returns them to the client.

    - Natural Language Querying (/query)
        - The /query endpoint first uses the Google Gemini model to convert the user's natural language query into structured queries for each database. It then calls query_mysql() for MySQL, query_mongodb() for MongoDB, and query_firebase() for Firebase, passing the generated queries to each connector.
        - The results from each database are collected, and if possible, merged based on shared identifiers (like listing IDs) before being returned to the client. The endpoint logic ensures that only the relevant databases are queried, depending on the user's request and the data fields involved.
    - Data Modification (/modify)
        - The /modify endpoint processes data modification requests (insert, update, delete) in a similar fashion. It uses Gemini to convert the user's natural language modification command into structured modification instructions for each database.
        - Then, it calls modify_mysql() for MySQL, modify_mongodb() for MongoDB, and modify_firebase() for Firebase, passing the generated instructions to each connector. Each connector executes the modification and returns a status or result, which the endpoint collects and returns to the client.
- Frontend UI (App.js)
    - Operation Mode Selection and Database Selection
        - Query: Execute natural language queries on MySQL, MongoDB, or Firebase.
        - Explore Schema: View tables, collections, structures, and sample data.
        - Modify Data: Perform insert, update, or delete operations.
        - A dropdown selector allows users to specify the target database (MySQL / MongoDB / Firebase).
        - The selected database type is sent to the backend along with the query/modification request.
    - Natural Language Input & Example Queries
        - Users input their questions in a text field
        - Predefined example queries are shown based on the selected operation to guide users and simplify testing.
    - Request Handling with Axios

  - Axios sends POST requests to backend endpoints (/query, /explore, /modify).
  - Loading state is handled during request processing, and error messages are shown to users.
  - After a response is received, the UI displays both the converted query and raw results for showing and debugging.
- Dynamic Result Rendering
  - <DataTable />: Displays structured query results from SQL or NoSQL databases.
  - <ListDisplay />: Lists tables, collections, or node paths.
  - Users can choose the visibility of converted queries and raw output using expandable panels.
- Component-Based Architecture
  - Display components are reusable, simplifying future support for new databases or data formats.
- User Experience Enhancements
  - Responsive design using Material UI components
  - Button states, query clearing, and query examples improve interaction usability.

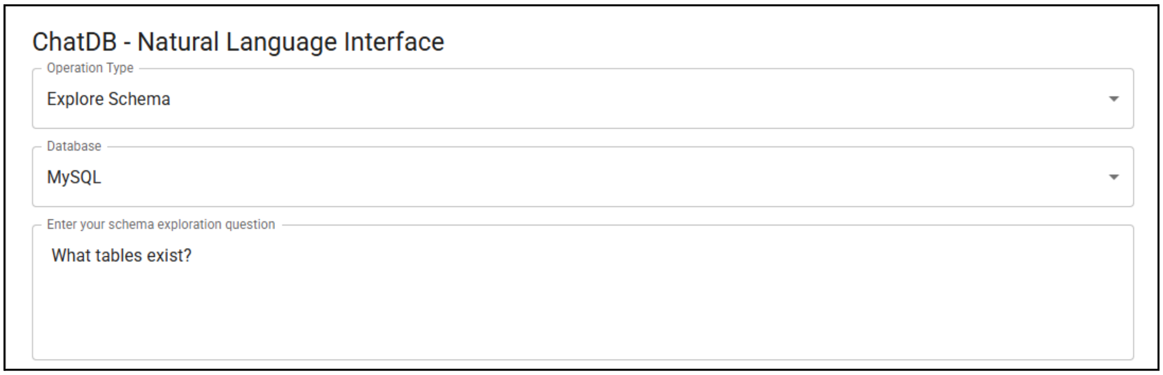
Tech Stack

- Core Language
  - Python 3
- Data Processing
  - Pandas: used in data integration scripts to read, clean, and preprocess CSV data before loading into databases.
  - NumPy: handling numeric conversions and NaN detection (e.g., in the Mongo connector to translate NaN to None).
  - AST: Safe parsing of string-encoded lists (amenities) in the Mongo loader.
- Database Connectors
  - MySQL (PyMySQL): handles querying, schema validation, and data modification for the MySQL database.
  - MongoDB (PyMongo): handles querying, schema exploration, BSON-to-JSON conversion, and data modification for MongoDB.
  - Firebase (firebase_admin): handles querying, schema exploration, and data modification for Firebase Realtime Database.
- Web Framework & API Layer

- - FastAPI (FastAPI, HTTPException): provides the main REST API endpoints (/explore, /query, /modify), request validation, and error handling.
  - Uvicorn (uvicorn): ASGI server used to run the FastAPI application.
  - Pydantic (BaseModel): defines request models (QueryRequest, ExploreRequest, ModificationRequest) for input validation and automatic documentation.
  - CORS Middleware: enables Cross-Origin Resource Sharing, allowing the frontend (possibly on a different domain/port) to access the backend API.
- Frontend
  - React.js: main JavaScript library for building the user interface as a single-page application (SPA). Handles state, component rendering, and user interactions.
  - Material UI (MUI): Provides pre-built, customizable UI components for a modern and responsive design.
  - Axios: Handles HTTP requests to the backend API endpoints (/query, /explore, /modify).
  - React Hooks (useState): Manages component state (e.g., query text, results, loading status, selected database, etc.).
  - Material-UI ThemeProvider (ThemeProvider, createTheme): Provides consistent theming and color palettes across the application.
- Natural-Language Processing
  - Google GenAI (google.genai): Converts natural language queries and modification commands into structured queries or instructions for each database.

Implementation Screenshots

Schema Exploration

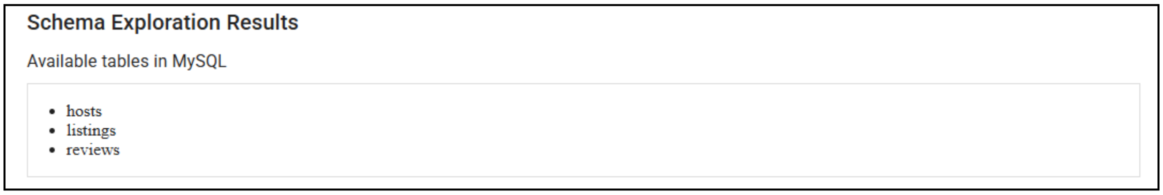
- 

  ChatDB - Natural Language Interface

  Operation Type: Explore Schema

  Database: MySQL

  Enter your schema exploration question: What tables exist?

- 

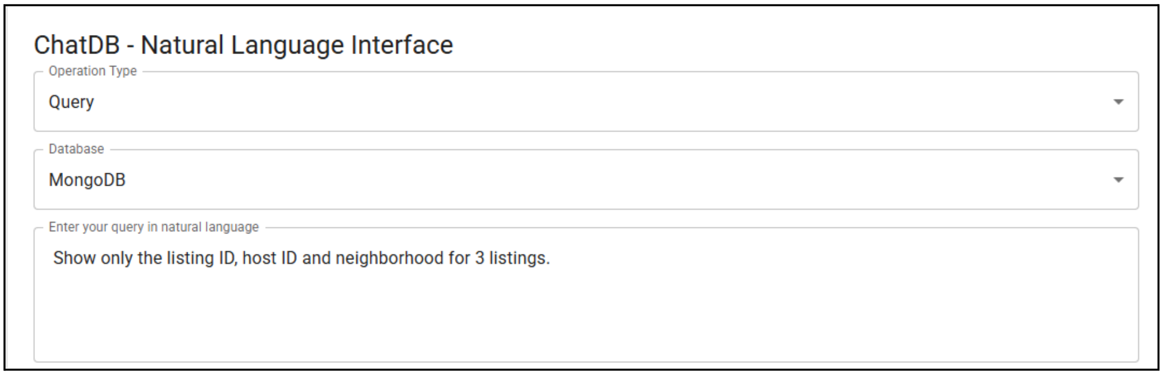
  Schema Exploration Results

  Available tables in MySQL

  - hosts
  - listings
  - reviews

  - This function allows the user to quickly view the database structure through natural language: select the "Explore Schema" operation on the interface, then select the target database (MySQL is selected in the Screenshot), enter a question such as "What tables exist?", and the backend will automatically perform metadata queries and list all table names (such as hosts, listings, reviews).

Database Query

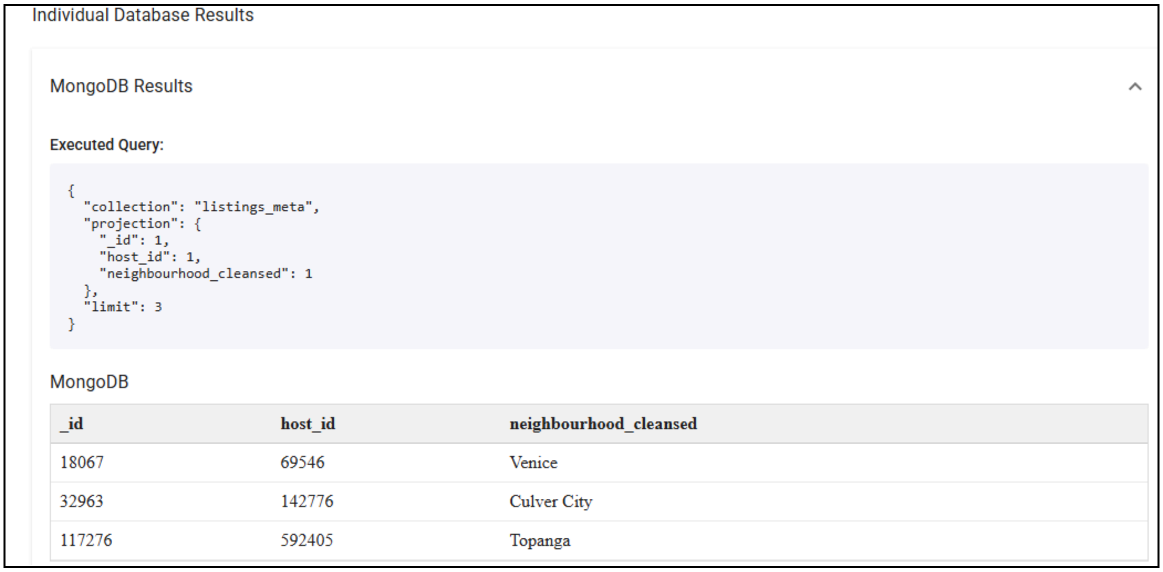
- 

  ChatDB - Natural Language Interface

  Operation Type: Query

  Database: MongoDB

  Enter your query in natural language: Show only the listing ID, host ID and neighborhood for 3 listings.

- 

Individual Database Results

MongoDB Results

Executed Query:

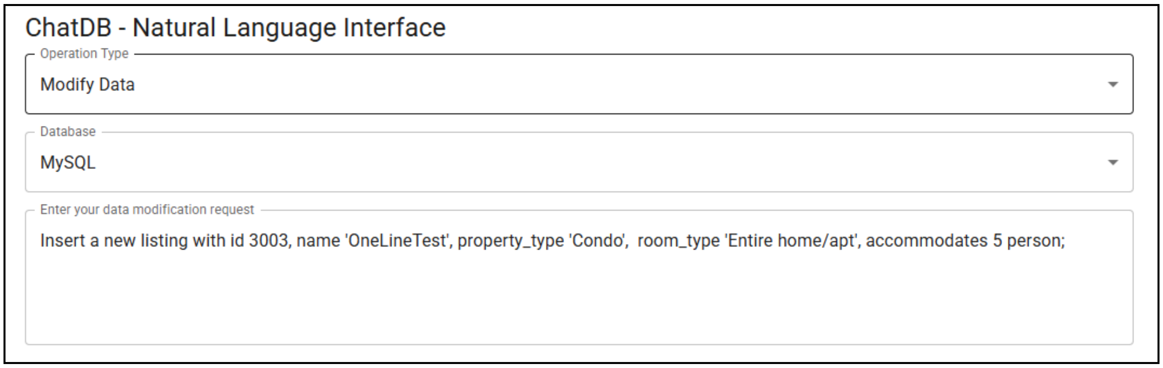
```
{
  "collection": "listings_meta",
  "projection": {
    "_id": 1,
    "host_id": 1,
    "neighbourhood_cleansed": 1
  },
  "limit": 3
}
```

MongoDB

_id	host_id	neighbourhood_cleansed
18067	69546	Venice
32963	142776	Culver City
117276	592405	Topanga

  - This function allows the user to query database content through natural language: select the "Query" operation on the interface, select the target database (MongoDB is selected in the screenshot), enter a natural language query description like "Show only the listing ID, host ID and neighborhood for 3 listings.", and the backend will automatically generate and execute the corresponding MongoDB query statement, and present the executed original query and the returned table results together.

Modification operation

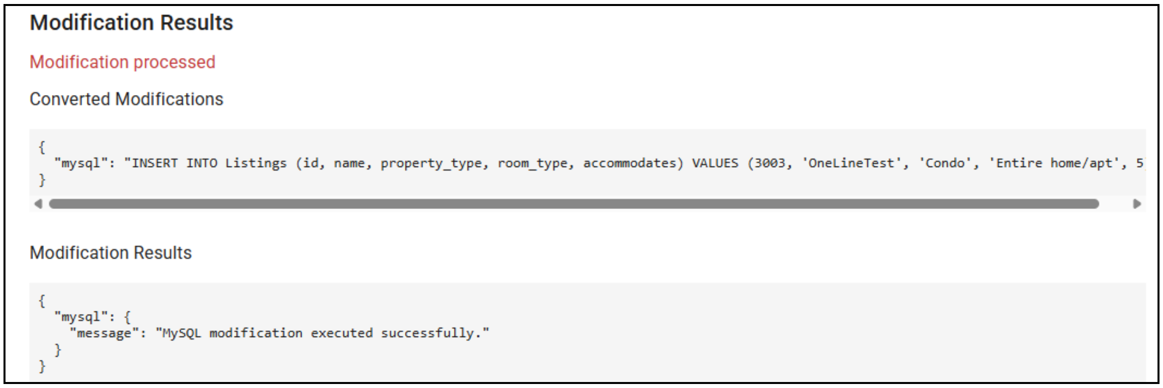
- MySQL Example:
- 

ChatDB - Natural Language Interface

Operation Type: Modify Data

Database: MySQL

Enter your data modification request: Insert a new listing with id 3003, name 'OneLineTest', property_type 'Condo', room_type 'Entire home/apt', accommodates 5 person;

- 

Modification Results

Modification processed

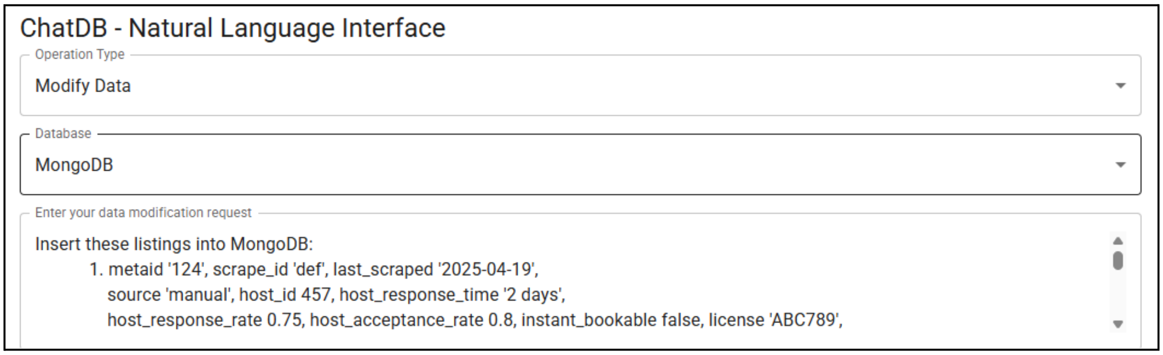
Converted Modifications

```
{
  "mysql": "INSERT INTO Listings (id, name, property_type, room_type, accommodates) VALUES (3003, 'OneLineTest', 'Condo', 'Entire home/apt', 5
}
```

Modification Results

```
{
  "mysql": {
    "message": "MySQL modification executed successfully."
  }
}
```

  - This is an example of inserting new data into a MySQL database, inserting only one record. After the user enters natural language in the UI interface, the backend will generate the corresponding SQL insert statement, execute the statement, and return a successful execution result to the UI interface.

- MongoDB example:
- 

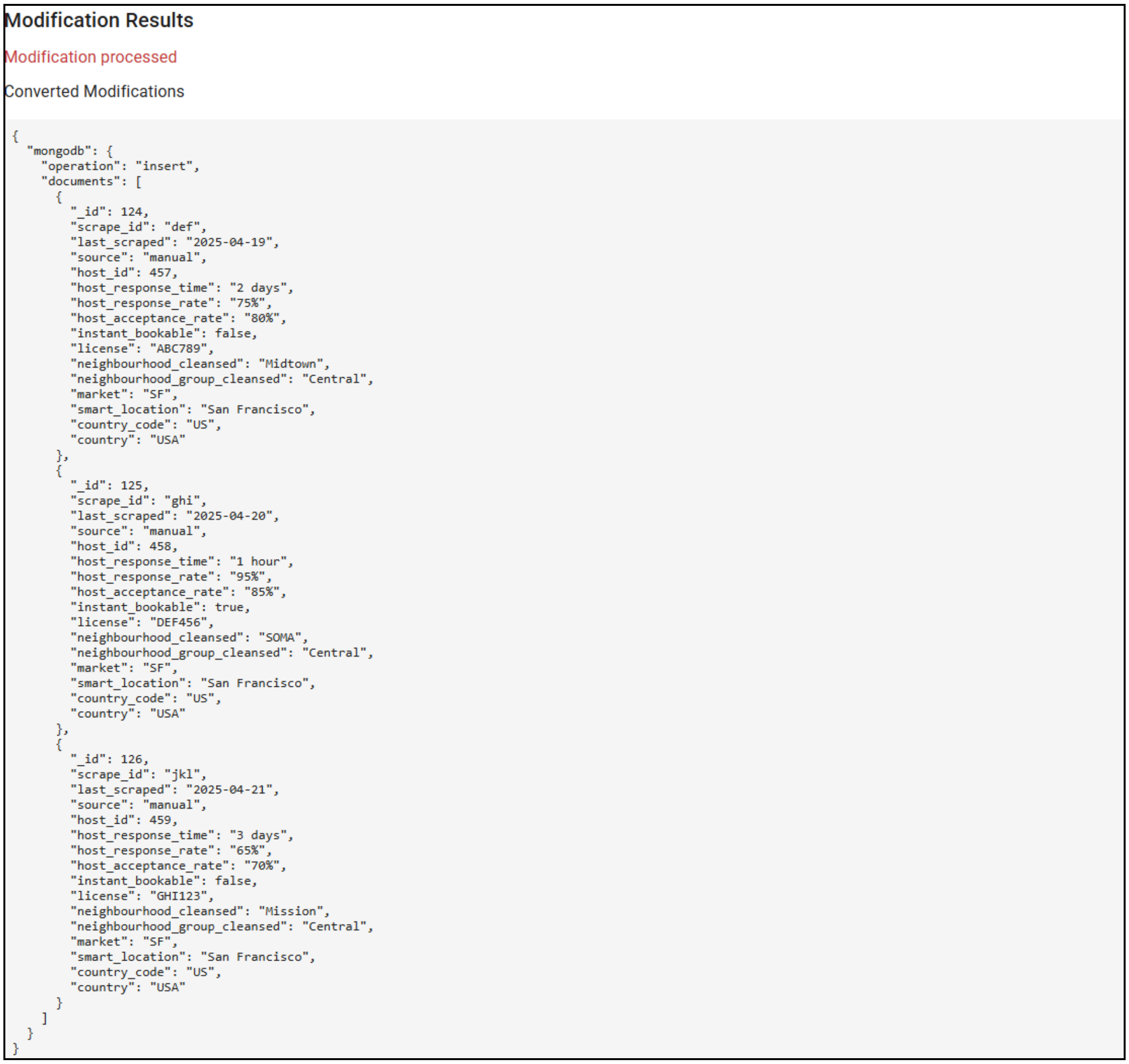
ChatDB - Natural Language Interface

Operation Type: Modify Data

Database: MongoDB

Enter your data modification request

Insert these listings into MongoDB:
1. metaid '124', scrape_id 'def', last_scraped '2025-04-19',
source 'manual', host_id 457, host_response_time '2 days',
host_response_rate 0.75, host_acceptance_rate 0.8, instant_bookable false, license 'ABC789',

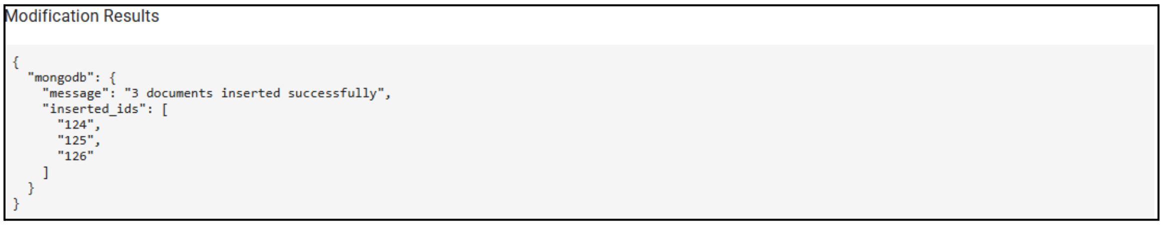
- 

Modification Results

Modification processed

Converted Modifications

```
{
  "mongodb": {
    "operation": "insert",
    "documents": [
      {
        "_id": 124,
        "scrape_id": "def",
        "last_scraped": "2025-04-19",
        "source": "manual",
        "host_id": 457,
        "host_response_time": "2 days",
        "host_response_rate": "75%",
        "host_acceptance_rate": "80%",
        "instant_bookable": false,
        "license": "ABC789",
        "neighbourhood_cleansed": "Midtown",
        "neighbourhood_group_cleansed": "Central",
        "market": "SF",
        "smart_location": "San Francisco",
        "country_code": "US",
        "country": "USA"
      },
      {
        "_id": 125,
        "scrape_id": "ghi",
        "last_scraped": "2025-04-20",
        "source": "manual",
        "host_id": 458,
        "host_response_time": "1 hour",
        "host_response_rate": "95%",
        "host_acceptance_rate": "85%",
        "instant_bookable": true,
        "license": "DEF456",
        "neighbourhood_cleansed": "SOMA",
        "neighbourhood_group_cleansed": "Central",
        "market": "SF",
        "smart_location": "San Francisco",
        "country_code": "US",
        "country": "USA"
      },
      {
        "_id": 126,
        "scrape_id": "jkl",
        "last_scraped": "2025-04-21",
        "source": "manual",
        "host_id": 459,
        "host_response_time": "3 days",
        "host_response_rate": "65%",
        "host_acceptance_rate": "70%",
        "instant_bookable": false,
        "license": "GHI123",
        "neighbourhood_cleansed": "Mission",
        "neighbourhood_group_cleansed": "Central",
        "market": "SF",
        "smart_location": "San Francisco",
        "country_code": "US",
        "country": "USA"
      }
    ]
  }
}
```

- 

Modification Results

```
{
  "mongodb": {
    "message": "3 documents inserted successfully",
    "inserted_ids": [
      "124",
      "125",
      "126"
    ]
  }
}
```

  - This is an example of batch inserting new data into a MongoDB database. The user describes the three records to be inserted in natural language in the UI interface, and the backend will convert it into the corresponding insertMany operation. After executing the command, the successful execution message will be returned to the UI.

Learning Outcomes

- Multi-Database Integration
    - By integrating MySQL, MongoDB, and Firebase into the system simultaneously, we practiced our ability to work with different database models (relational, document, and NoSQL tree structures), including query language syntax, connection methods, and processing mechanisms among different databases.
- Natural Language Processing for Databases and LLM Function Principles
    - In the project, we used large language models to convert the natural language input by users into SQL/NoSQL query statements; specifically, we explored how to convert natural language queries into database-specific queries for MySQL (SQL), MongoDB (JSON-based queries/aggregations), and Firebase (NoSQL path-based queries) based on carefully engineered prompts for LLMs (Google Gemini) to generate valid, structured queries and modifications from user input.
    - During the tuning process of formatting the LLM prompt, we noticed how prompt clarity, specificity, and examples affect the quality and reliability of LLM outputs. Built upon how LLMs process context, constraints, and formatting requirements, we learnt to handle the limitations and strengths of LLMs when generating code for different data models and query languages.
- Full-Stack Application Development, Collaboration, and Modularity
    - By separating the frontend and backend through Flask and React, we gained solid practice experience on designing and implementing a modular, collaborative system where the backend and frontend are clearly separated, communicate through well-defined APIs, and can be developed and maintained independently. The collaboration process enables parallel teamwork, easier debugging, and extensibility, which are the key skills for working effectively in real-world software projects.
    - In addition, this project covered multiple dimensions such as architecture design, interface definition, database configuration, exception handling, performance optimization, and exercised the division of labor and collaboration as well as comprehensive development capabilities among each group member.

Challenges Faced

- The accuracy issue of natural language query conversion
  - Since Users may phrase queries in many ways, when using large language models to convert the natural language input by users into SQL or NoSQL queries, there exist problems such as semantic misunderstanding, field recognition errors, and syntax that does not conform to the database structure.
  - To address it, we are aware that the LLM must fully understand the schema differences (e.g., price only in Firebase, not in MySQL/MongoDB); Prompts to Google Gemini include detailed schema descriptions and explicit instructions for each database, ensuring the LLM understands field availability and query structure.
  - We also implemented a validation process that attempts multiple times to parse and validate the LLM's output as JSON, using fallback logic if the output is invalid. The code checks for the presence of required keys and only proceeds if all are present and valid. If the LLM fails to produce valid output after several attempts, the system logs the error and returns an empty result, preventing crashes.
- The unified interface design of different databases is complex
  - The three databases (MySQL, MongoDB, and Firebase) have significant differences in data models, connection methods, and syntax formats. Designing a unified query scheduling and result parsing interface requires a lot of compatibility processing.
  - We implemented several solutions, such as defining helper functions like query_mysql, query_mongodb, and query_firebase abstract away the differences in query execution and result formatting. The backend logic determines which databases to query based on the user's request and the LLM's output, avoiding unnecessary or invalid queries, such as routing field-specific queries (eg, Price data only exists in Firebase) to the corresponding database.
- The React framework UI needs to maintain the complex state management
  - The frontend needs to render various result formats based on the returned structure of the backend. The returned structures of different databases are inconsistent, and the field parsing and display logic need to be frequently adjusted during the debugging process.
  - To address that, the backend is improved to return structured responses with clear messages, data, and error fields, making it easier for the frontend to display results and handle errors. Besides, we added a dedicated endpoint for schema exploration and sample data (/explore) helps the frontend manage and display database metadata and samples separately from query results.

- Database connection and permission management issues
  - During the development, the connection configuration is relatively complex, involving identity authentication, cross-domain settings, and permission control. We spent more time on server-side configuration and the debugging of query permissions.
  - In order to manage the databases, each database connector (mysql_connector, mongodb_connector, firebase_connector) handles its own connection logic, pooling, and error handling, which makes it easier to locate and fix issues without affecting other parts of the system. If a database schema or credential changes, we only need to update the corresponding connector, reducing the risk of introducing bugs elsewhere.

Individual Contributions

- Yijian Jin:
  - Responsible for backend system design and data integration infrastructure using FastAPI, including data ingestion pipelines and connector module configuration. Implemented endpoints, integrated the Google Gemini API for query translation, and composed unified query functions across all databases. Conducted functionality testing and optimized query workflows for accuracy and performance. Contributed to the final report regarding the backend implementation, challenges faced, and future scope sections.
- Hongyu Yu :
  - Responsible for backend modification module development: built FastAPI interfaces to translate natural-language “Modify Data” requests into SQL/MongoDB commands (insert/insertMany, update/updateMany, delete/deleteMany), and also performed end-to-end testing to locate and fix various bugs in the backend modification process. Contributed to the final report by completing the implementation screenshots section and checking the functionalities part to identify and fill in any omissions.
- Zilu Wang :
  - Responsible for frontend system design and full implementation using React, including component architecture, user interaction logic, and UI responsiveness. Led frontend-backend integration, handled endpoint testing, and validated end-to-end data flow. Also conducted partial and overall system testing to ensure stability. Contributed to the final project report by drafting the architecture design, frontend implementation, and learning outcome sections.

Conclusion

This project successfully constructed a multi-database system, ChatDB, for natural language queries(CRUD), enabling users to uniformly access and operate the three databases of MySQL, MongoDB, and Firebase through natural language. The frontend of the system provides an intuitive query interface. The backend uses large language(Google Gemini) models to translate natural language into structured query statements, automatically routes to the corresponding database processing results according to the query type, and finally returns structured display. Overall, this project demonstrates the performance of combining modern AI with software engineering to bridge the gap between human language and complex data management systems.

Future Scope

- Improve the parsing accuracy of LLM
    - At present, the performance of LLM depends on context and Prompt design. In the future, its parsing accuracy can be improved and incorrect queries can be reduced through Fine-tuning, domain knowledge injection, or RAG/MCP mechanisms; also, we can implement more advanced post-processing to validate and correct LLM-generated queries before execution.
- Strengthen query security and permission control
    - During the demo of other groups' projects, we noticed that some groups use user authentication and authorization login management modules, which may prevent malicious queries or illegal operations and enhance system security. Based on that, we could add user authentication and role-based access control to restrict who can query or modify which data.
- Optimize performance and history caching
    - Introduce paging mechanisms and cache optimization for large-scale queries. Use specific data structures to do history caching(like LRU, LFU).
- Introduce a visual UI query builder
    - Provide a graphical interface to help users construct query statements and decrease keyboard input to increase efficiency. Show interactive diagrams of tables, collections, and relationships to help users understand the data model.