Final Report: ChatDB for Pokémon Card Trading Market

Team Details:

Zixi Wang (UID: 2854187591)

1. Introduction

The goal of this project is to develop a natural language interface that enables users to query MongoDB databases for Pokémon cards and related media content using plain English. The system allows users to ask English questions (e.g., "Show me all Fire-type Pokémon cards") and automatically translates them into executable MongoDB queries, supporting operations like find, aggregate, $lookup, insert, update, and delete. This eliminates the need for users to understand the query language or the database language.

2. Planned Implementation (From Project Proposal)

Originally, this project aimed to create a natural language query interface (NLI) for interacting with both SQL (MySQL) and NoSQL (MongoDB) databases, allowing retrieval, analysis, and modification of Pokémon card market data. However, given the scope and the solo nature of the work, the implementation was adapted mid-project to focus solely on MongoDB.

Initial Plan Highlights (from Proposal):

- **Use MySQL for structured card attributes like name, rarity, price, and condition.**
- **Use MongoDB for unstructured data like transaction history and user behavior.**
- **Support natural language queries such as:**
  - **"Find all Charizard cards"**
  - **"List all cards under $50"**
  - **"Update the price of Pikachu to $80"**
- **Provide a web interface with input box and result visualization.**
- **Use OpenAI API to translate English to SQL or MongoDB queries.**

Changes & Focus:

- **Dropped SQL integration due to limited time and complexity of handling both systems.**
- **Prioritized MongoDB only to focus on:**
  - **Robust data model with 8 collections.**
  - **Complex join logic using $lookup and $expr.**

    - **Read and write operations (find, aggregate, insertOne, updateOne, deleteOne).**
- **Deferred full Web UI implementation; focused on terminal interface using Python.**
- **Integrated natural language query system powered by OpenAI GPT, using a carefully engineered prompt that encodes schema, relationships, and query rules.**

This shift allowed deeper focus on prompt engineering, MongoDB aggregation logic, and multi-collection querying while keeping the project scope feasible and technically valuable.

3. Architecture Design

Flow Diagram

User Input(English)
→ app.py (Main controller)
→ nlp_transfer.py (OpenAI-based prompt logic)
→ app.py (receive transferred query then send and send to mongodb.py)
→ mongodb.py (Execute the returned MongoDB query)
→ MongoDB
→ app.py (Formatted output)

This modular architecture separates concerns clearly between user interface, query generation, database access, and data preparation.

Description

- **app.py handles user interaction and orchestrates the query lifecycle which handles user input, manages control flow, and outputs results.**
- **nlp_transfer.py constructs a detailed prompt with schema and rules, and invokes OpenAI's GPT model to generate the query.**
- **mongodb.py executes the query using pymongo, and returns formatted results, it executes operations such as find, aggregate, insert, update, and delete.**
- **MongoDB stores all relevant collections.**
- **data_upload.ipynb: A separate Jupyter Notebook was used to clean, transform, and upload data into the MongoDB database.**

4. Implementation

1. Functionalities

My system supports a wide range of query types. For basic queries, I implemented `find()` with projection, skip, and limit, so users can get only the fields they need, and avoid loading thousands of rows at once. For more complex tasks, I used `aggregate()`

with stages like `$match`, `$group`, `$sort`, `$limit`, `$count`, `$project`, `$unwind`, and `$lookup`.

I also supported natural joins across two or even three collections using `$lookup` with `$expr`, and tested insertOne, insertMany, updateOne, and deleteOne operations. Users can also count, filter, and search schema-related information like all available Pokémon types or generations.

Altogether, I tested 15 queries that cover different levels of complexity.

2. Tech Stack

Python: I used Python for the whole backend and controller logic.
MongoDB: My main database, where I stored all 8 collections.
OpenAI GPT-3.5: I used this model to convert natural language to MongoDB operations.
BeautifulSoup & Requests: These helped me scrape real-time Pokémon card price data from eBay.
Jupyter Notebook: I used it for cleaning, modifying, and uploading data to MongoDB.

3. Implementation Details & Screenshots

I created 8 collections, each representing a different part of the Pokémon world:

1. **`card_informations` — card details like name, type, HP, etc. (18,688 rows)**
2. **`card_price` — scraped price data from eBay (14,290 rows)**
3. **`card_sets` — info about each card set and its generation (164 rows)**
4. **`deck_cards` — which cards appear in which decks (3,976 rows)**
5. **`pokemon_decks` — metadata about decks (188 rows)**
6. **`pokemon_game` — games by generation and platform (58 rows)**
7. **`pokemon_movies` — movie metadata (22 rows)**
8. **`pokemon_information` — base stats and evolution (809 rows)**

The collections in my MongoDB database are not isolated — they are connected through shared fields that allow for multi-collection queries using $lookup. Here's how they relate:

- **card_informations.name matches card_price.name, and**
- **card_price.set_id matches card_sets.id.**
  Together, these two fields let me join card details with their market prices and set information. In fact, card_price acts as a "relational table" that bridges card_informations and card_sets.
- **deck_cards.deck_id links to pokemon_decks.id, and**

**- deck_cards.name connects to card_informations.name.
 This means deck_cards plays the role of a many-to-many relationship table between card_informations and pokemon_decks.**
**- pokemon_game.generation, pokemon_movies.generation, and card_sets.generation are all aligned.
 This allows me to connect games, movies, and card sets by their generation label.
 These three collections form a many-to-many relationship structure — for example, one generation of games might relate to multiple movies and sets.**
**- pokemon_information.name connects to both card_informations.name and deck_cards.name, forming a one-to-many relationship.
 That is, one Pokémon (like Pikachu) can have many card versions and appear in many decks.**

These connections are essential for implementing $lookup joins in my query system and make the data model both flexible and realistic for exploring the Pokémon card ecosystem.

How I Built My Dataset:

- I took most of the time building my dataset. I first collected card and set data from websites like Bulbapedia, then wrote Python code to open the JSON files and load the data into MongoDB.
- For prices, I wanted real-world values. So I used BeautifulSoup to scrape eBay. Since eBay blocks too many fast requests, I had to slow down and scrape carefully. It took me around 10 hours to finish, but I ended up with over 14,000 accurate price records.
- For the game and movie data, I used Wikipedia, saved the information into JSON manually, and then uploaded it using Python.

Output Design:
- I tried to make the output clean and readable. For example:
- In \`find\` queries, I only show the top 3 documents, then tell the user how many total results.
- For \`insert\`, I show the last 3 inserted rows so the user knows they were added.
- For \`update\`, I show only the updated row.
- For \`delete\`, I show both before and after, to highlight what was removed.
- I also print out the equivalent MongoDB shell command so users can learn how the query works.

I took six screenshots of my query tests. Each folder focuses on one kind of query: basic \`find\`, projection, \`$lookup\` joins, groupings, and also \`insert\`, \`update\`, and \`delete\`. This way, it's easier to organize and understand.

implementation_path is https://drive.google.com/drive/folders/1jiSowkxaZilKIwyNVhW3xC7wa4MweuGa?usp=drive_link

Overall, I wanted the system to feel simple but powerful — and the output should help people understand what happened, even if they don't know MongoDB.

5. Learning Outcomes

This project helped me learn a lot, both technically and conceptually. First, I gained a deep understanding of how to construct complex MongoDB aggregation pipelines, especially those involving nested joins with `$lookup`, `$unwind`, and `$project`. I also learned the key difference between `find()` and `aggregate()`—not just in syntax, but in flexibility, performance, and use cases.

This project also gave me my first exposure to frontend concepts. Even though I only used a terminal interface, I now understand how the frontend and backend can be connected, and I'm more confident to try tools like Flask in the future. I also became more comfortable with data scraping. By building my own crawler for eBay prices, I improved my data mining skills and learned how to handle real-world web data.

Another major learning outcome was in the area of prompt engineering. I had to carefully design a multi-part prompt that described my entire database schema, field relationships, and example use cases. This taught me how to guide a large language model to generate reliable, structured NoSQL queries. I also practiced clean modular design and data abstraction, using multiple Python files with clear responsibilities. Finally, I learned how to validate data types (like converting price strings into floats), and how to format outputs for better user readability.

6. Challenges Faced

One of the biggest challenges I faced was ensuring that the joins between collections worked correctly. Many Pokémon cards have the same name but appear in different sets, so I had to use composite keys like `name + set_id` and use `$expr` conditions in `$lookup`. Because my raw data was large, messy, and inconsistent, my first step was to do careful data cleaning. After performing the joins, the connections were not always obvious, so I had to manually verify the results. I renamed many fields and added new ones to make joins possible—for example, I added a custom `generation` attribute to movies and games, and I also modified the names of my sets so that they could match up correctly with card information.

The second challenge was finding the right datasets. At first, I only found three Pokémon card datasets and didn't fully understand the project requirements. I later realized that to

support multi-table joins, I needed collections that had many-to-many relationships. So I spent a lot of time searching for more useful data. I eventually added pricing data and a set-card link to form a bridge between card and set data. I also created custom datasets for Pokémon movies and games to enrich my database. Linking all of these collections together became the most time-consuming and difficult part of the entire project.

Another major challenge was finding up-to-date price data for the cards. Card prices change frequently and I couldn't find any public dataset that covered tens of thousands of recent cards. In the end, I decided to scrape prices directly from eBay. This was very difficult because of the large volume—I spent around 10 hours scraping, and I had to slow down the request speed to avoid being blocked. I even added a counter system so that if the script failed halfway through, I could resume from the last point without starting over.

Finally, I faced challenges with the OpenAI API. I initially tried using free/open models like Llama 2 and Mistral 7B via Hugging Face, but they either didn't support MongoDB structure well or crashed due to memory limits. No matter how I tuned the prompt, they couldn't generate a usable query. So I gave up and switched to GPT. Even with GPT, I still had to write detailed, structured prompts to guide it. Sometimes, the model would still return malformed JSON or hallucinated fields, so I had to fine-tune the instructions again and again to make it stable and reliable.

7. Individual Contribution

I was responsible for everything from design to deployment. I designed the overall schema of the database and collected the datasets from various sources like github, eBay, and Wikipedia. For real-time card pricing, I wrote a web scraper using BeautifulSoup to collect prices from eBay, which took several hours due to anti-scraping limits. I wrote all the Python modules, including `app.py` for user interaction, `mongodb.py` for database operations, and `nlp_transfer.py` for prompt-based query generation.

I also developed the full prompt used for GPT-based translation, and tested it with many edge cases. I created the Jupyter Notebook `data_upload.ipynb` to handle data cleaning and batch upload. In addition, I prepared the project demo, README, and final presentation slides.

8. Conclusion

This project successfully fulfills all the functional and structural requirements of a NoSQL-based natural language query system. From data collection to database construction, from prompt design to system implementation, every component works together to allow users to explore, retrieve, and modify complex Pokémon-related data using plain English. The system supports real-world use cases such as searching for

cards by type or HP, looking up card prices, joining across multiple collections like cards and sets, and even updating or inserting new data—all powered by a flexible MongoDB backend.

One of the biggest highlights of this project is how it demonstrates the integration of large language models with database query logic. The system not only responds to natural language input but also interprets the intent and generates structured queries behind the scenes. This greatly lowers the barrier for non-technical users to interact with complex data systems.

From a personal perspective, I'm proud of how far this project pushed my technical abilities. I went from having no experience in building end-to-end user-query-database systems to implementing a modular, functional tool that is both practical and extensible. I also gained confidence in working with messy real-world data, applying prompt engineering, and designing joins that make sense across multiple domains like cards, games, and movies.

Most importantly, this project gave me a real sense of how AI and databases can work together—not just as two separate fields, but as a combined system that can serve people in new and intelligent ways. The experience has made me more excited to keep exploring the connection between NLP, data systems, and user experience design.

9. Future Scope

There are many ways this project can be extended. Certainly, the first improvement is to add support for relational databases like PostgreSQL, so the system can switch between SQL and NoSQL depending on the query. Another improvement is to build a web-based frontend using Flask or React, so users can interact with the system visually instead of using a terminal.

Another important direction is to enhance the price tracking feature. The current price data is static and basic, but in the future it could be made dynamic by automatically updating prices on a regular basis. This would allow users to check real-time prices of different Pokémon cards, compare cards across sets, and even view price histories.

Other useful improvements would be adding more visual elements. For example, displaying card images or Pokémon pictures in the user interface would make the system more intuitive and engaging. Charts and graphs—such as average stats by type or card price distributions—could also help users better understand the data. The system structure can also be improved and made more organized for easier maintenance and expansion. These updates would make the project more practical and enjoyable for both casual fans and serious collectors.

Included Artifacts

My final report folder link is https://drive.google.com/drive/folders/1REMPq-IA7pmZ4wyQKgA-MygrNEzKkPUX?usp=drive_link

inside my folder I have several files:

- **Sample Output folder: include all my sample output**
- **exported_full_data folder: include all my data .json files**
- **_pycache_ folder: the progress to run the app.py file**
- **and all my py and .ipynb files, include app.py, nlp_transfer.py, mongodb.py and data_upload.ipynb**

I also include:

- **Dsci-551 presentation.pdf: it is my presentation slide for demo**
- **sample_data.json: include all my data sample which only include top five examples**
- **README.md: includes step by step guide as to how to implement my code**
- **requirements.txt: virtual environment with all the necessary libraries**

API Key Warning

The OpenAI API is only used in a single module: nlp_transfer.py. In this file, the system calls the OpenAI GPT-3.5 API to convert natural language into MongoDB queries. The required credentials include the API_KEY, PROJECT_ID, and ORGANIZATION_ID.

All API credentials have been removed from the submitted code for security. Users must manually insert their own OpenAI credentials in the placeholders within nlp_transfer.py.

In contrast, the web scraping process in data_upload.ipynb uses public URLs and BeautifulSoup to collect real-time card prices from eBay. No additional APIs were used for that part.