4.2 Design Exploration

In order to complete this project, our group needed to deeply explore potential designs and come up with a solution.

4.2.1 Design Decisions

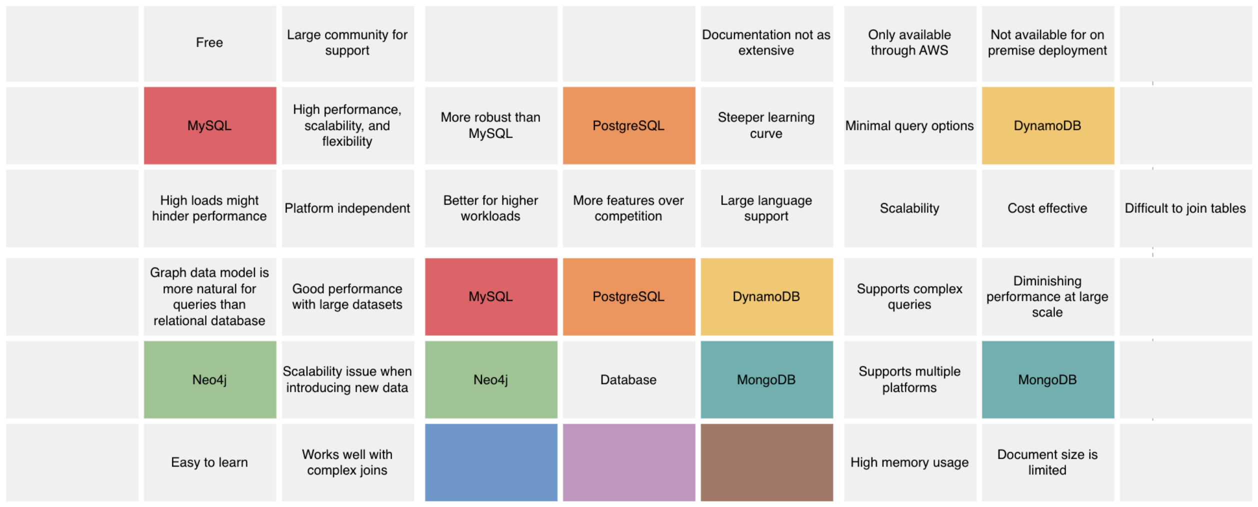
Due to the nature of our project, our design decisions consist primarily of software and technology selections. These include database management systems, user interface (UI) design, and generative AI model options.

One of the most important decisions we needed to make was which database management system we wanted to utilize. We decided on PostgreSQL for several reasons. The most important reason we chose this is due to it being a relational database. Given the large dataset, we decided a more structured database would better support retrieving relevant information. As opposed to other relational databases, however, PostgreSQL allows us to store data in custom data types. This will allow us to group aspects of data together for easier navigation and querying.

While most of our choices are purely for development and performance purposes, the UI design directly impacts the user experience. This makes it important for our design to be visually appealing and easy to use. As a result, we decided to implement our UI through the React library for TypeScript. While there are other similar libraries available, we chose to use React largely for its support of Virtual DOM (Domain Object Model). This allows us to quickly modify visual elements of the application as they need to be changed. Furthermore, React has a large user base and as a result, a large number of resources for speeding up development and enhancing applications.

We also needed to consider which AI language model would handle natural language processing. This leads us to choose OpenAI. We chose OpenAI due to the large number of tools that support our application and its sophisticated implementation. In particular, the LangChain library for Python supports generating SQL queries via OpenAI, allowing us to easily integrate our natural language processing into the rest of our application

4.2.2 Ideation



	Free	Large community for support			Documentation not as extensive	Only available through AWS	Not available for on premise deployment	
	MySQL	High performance, scalability, and flexibility	More robust than MySQL	PostgreSQL	Steeper learning curve	Minimal query options	DynamoDB	
	High loads might hinder performance	Platform independent	Better for higher workloads	More features over competition	Large language support	Scalability	Cost effective	Difficult to join tables
	Graph data model is more natural for queries than relational database	Good performance with large datasets	MySQL	PostgreSQL	DynamoDB	Supports complex queries	Diminishing performance at large scale	
	Neo4j	Scalability issue when introducing new data	Neo4j	Database	MongoDB	Supports multiple platforms	MongoDB	
	Easy to learn	Works well with complex joins	🟦	🟪	🟫	High memory usage	Document size is limited	

Figure 1: Ideation

4.2.3 Decision-Making and Trade-Off

Our decision-making process for design choices consisted of discussions amongst ourselves, personal knowledge, research, and input from our faculty advisor to compare and contrast the pros and cons of our options. When selecting a database system, we considered SQL and NoSQL database technologies and decided to use an SQL-based database. This was due to the fact that our data is heavily relational, consisting of many (~100) attributes linked to each star and stored in .csv format. This makes the querying functionality of an SQL database optimal for ease of use, speed, and storage of queries. Among the many options available for SQL database systems, we chose PostgreSQL because of its capability for flexible, complex querying, as well as the ability to create custom data types and not strictly limited to tables. As each star has many attributes that can be stored, PostgreSQL will enable us to manage our data much more efficiently through custom data types, such as information on star density.

4.3 Proposed Design

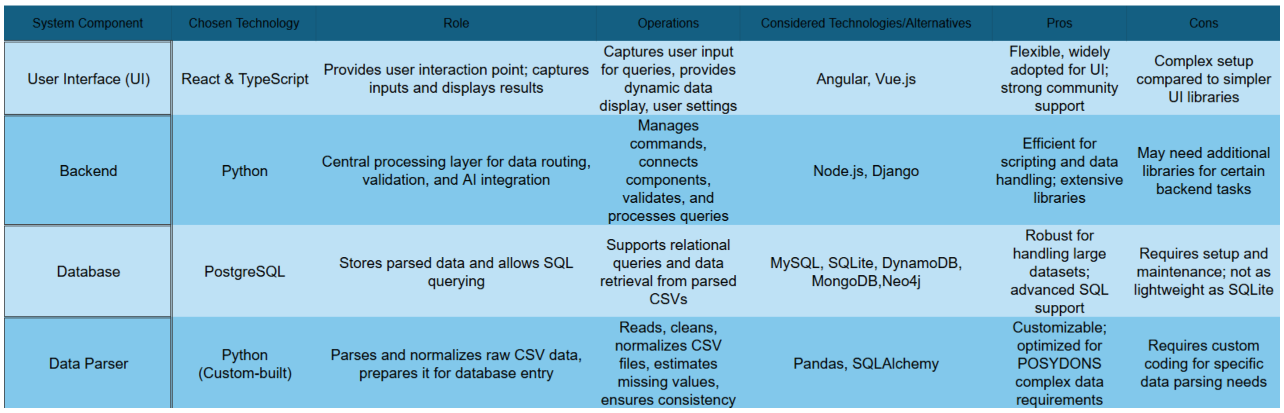
To implement our project, we came up with a detailed design consisting of deeply integrated subsystems.

4.3.1 Overview

Our project consists of four main subsystems. The user interface (UI), utilizing the React library for TypeScript, visualizes the application for the user. It allows the user to make natural language or SQL queries, displays queried data in an easy-to-under format and enables users to save their queries for later use. The UI is connected to the backend application. This system is responsible for communication between the UI and database as well as the mode of communication with OpenAI for handling natural language processing. When a user sends natural language to the backend, it sends this text alongside the database's blueprint to OpenAI. In response, OpenAI returns a properly structured SQL query. Before returning the query to the user, the backend determines if the query follows proper SQL syntax and retrieves data from the database. If the backend finds the query not valid, it makes a second attempt with OpenAI. If this still fails, it will return a message to the UI requesting the user to rewrite their message for clarity. If the query is deemed valid, however, it will be returned to the UI. At this point, the user can choose to accept or modify the query before returning it to the backend. The backend will then query the database. The database is a relational database managed by PostgreSQL. The database manages the data and returns the desired data upon retrieving a query from the backend. At this point, the backend sends this retrieved data to the user interface to be displayed. The final subsystem is the data parser. The data parser is a script that retrieves the CSV files generated by POSYDON and parses them. This parsed data is then stored in the database.

4.3.2 Detailed Design

Each technology within the system has been selected and integrated based on its unique capabilities to address specific functional and non-functional requirements. These choices support efficient data processing, user accessibility, and seamless interaction between the system's components. The following descriptions outline the purpose, role, and operations of each technology and component in achieving an optimized user experience, robust data management, and high query performance across extensive datasets. For ease of understanding, please refer to Figure 2.



System Component	Chosen Technology	Role	Operations	Considered Technologies/Alternatives	Pros	Cons
User Interface (UI)	React & TypeScript	Provides user interaction point; captures inputs and displays results	Captures user input for queries, provides dynamic data display, user settings	Angular, Vue.js	Flexible, widely adopted for UI; strong community support	Complex setup compared to simpler UI libraries
Backend	Python	Central processing layer for data routing, validation, and AI integration	Manages commands, connects components, validates, and processes queries	Node.js, Django	Efficient for scripting and data handling; extensive libraries	May need additional libraries for certain backend tasks
Database	PostgreSQL	Stores parsed data and allows SQL querying	Supports relational queries and data retrieval from parsed CSVs	MySQL, SQLite, DynamoDB, MongoDB,Neo4j	Robust for handling large datasets; advanced SQL support	Requires setup and maintenance; not as lightweight as SQLite
Data Parser	Python (Custom-built)	Parses and normalizes raw CSV data, prepares it for database entry	Reads, cleans, normalizes CSV files, estimates missing values, ensures consistency	Pandas, SQLAlchemy	Customizable; optimized for POSYDONS complex data requirements	Requires custom coding for specific data parsing needs

Figure 2: Technologies

4.3.2.1 High-Level Overview of the System:

User Interface (UI): Built using React and TypeScript, this layer provides users with interactive elements and visuals. It captures the user's input (e.g., SQL or NLP queries) and displays the query results, assisting in facilitating an intuitive end-user experience.

Backend: Manages requests from the UI, relays them to OpenAI for NLP processing, validates SQL requests and responses, and communicates with the PostgreSQL Database while also supporting the parsing and initialization of the database. The Backend acts as the bridge between the UI, database, and the AI model.

Database (PostgreSQL): The database system stores the data from the parsed and normalized CSV files and enables the SQL querying of the data.

Data Parser: Built using Python, the Data Parser Interprets the raw CSV files from POSYDON and normalizes them to make meaningful SQL queries possible. Reads the data from the CSV files and applies normalization methods to allow comparisons more readily between different simulations. Enter the data into the Database.

4.3.2.2 Detailed Subsystem Descriptions:

User Interface (UI) with React:

Role: Provides the primary user interaction point, enabling users to input SQL queries, view results, save, import database, and configure personal settings such as the ability to use OPENAI for NLP.

Operations: How React enables this, how it allows/displays the data dynamically, and works to provide this role and functionality.

Backend (Python):

Role: Serves as the central data processing layer, routing commands, validating queries, formatting responses, and the primary means of interconnecting components.

Operations: Connects the UI, database, and OpenAI, handling all data routing. Depending on whether the user is using natural language or raw SQL, the backend either directly processes the SQL query or uses OpenAI to generate a structured query. It validates all queries to ensure security and accuracy. Validated queries are executed, and the results are temporarily saved for the session before being sent back to the UI for display.

Database (PostgreSQL):

Role: Central data repository, where parsed CSV data from POSYDON is organized

Operations: Structured to handle relational queries, the database supports complex data types, allowing efficient data retrieval. The data relationships are designed to reflect the structure within the original CVS that was used to build the database.

Data Parser:

Role: Parses Raw POSYDON data, normalizes it, and enters it into the database

Operations: The parser, integrated with the backend, reads and interprets CSV files while standardizing data types. During this process, it performs initial data cleaning, including normalization, to ensure a uniform structure across entries. If gaps in the data are found, the parser estimates missing values by averaging similar local data points. This step is especially crucial given the massive datasets generated by POSYDON, which uses machine learning to simulate binary star systems. The parser ensures data integrity and accuracy, preparing all datasets for efficient storage and querying within PostgreSQL tables.

4.3.2.3 Visual Representation:

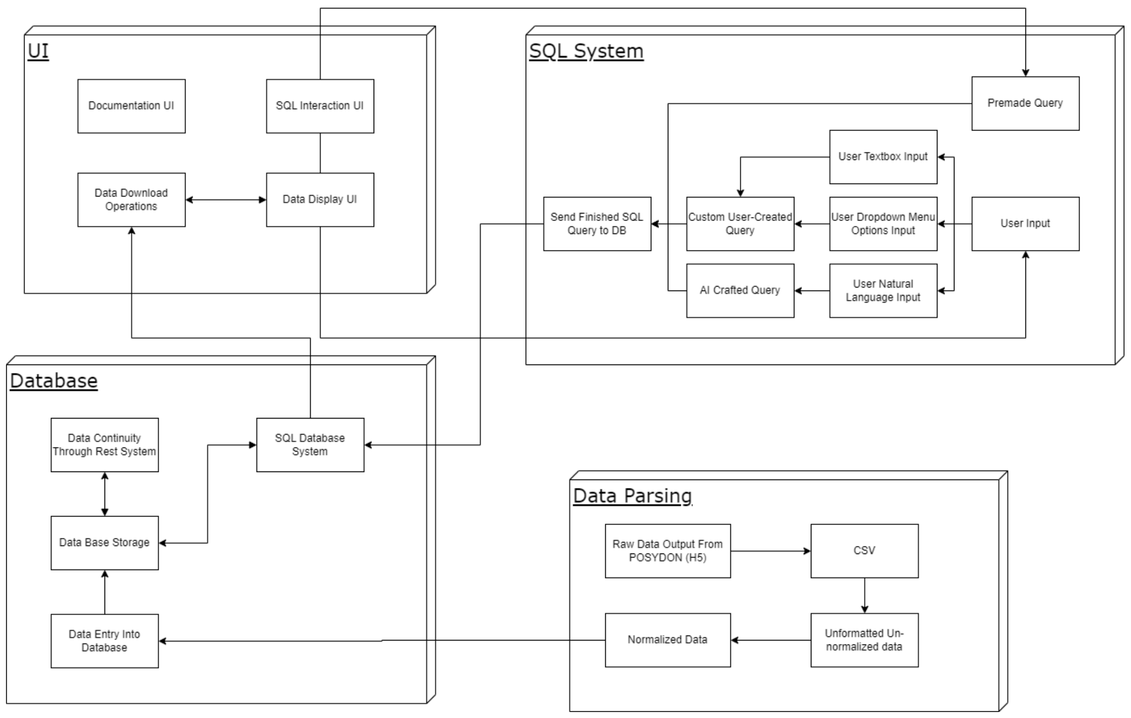


Figure 3: Visual representation

4.3.3 Functionality

In a typical use case of our application, an end user would use a local server or a high-spec personal computer to run a set-up script to host the database. The end-user would then run the desktop application used to parse the database, and running this, the application would also build the database and connect it to the application. After a successful connection, the end user would be able to perform two types of queries: a query using their own constructed SQL statement or an SQL statement generated through a machine learning-powered natural language processor, which would translate their request to a properly formatted SQL query. This query would then be performed, and a small subset of the data would be returned to the home screen of the application for preview. The whole result would then be available for download to a file on the end user's machine. The user could then perform another query or close the application.

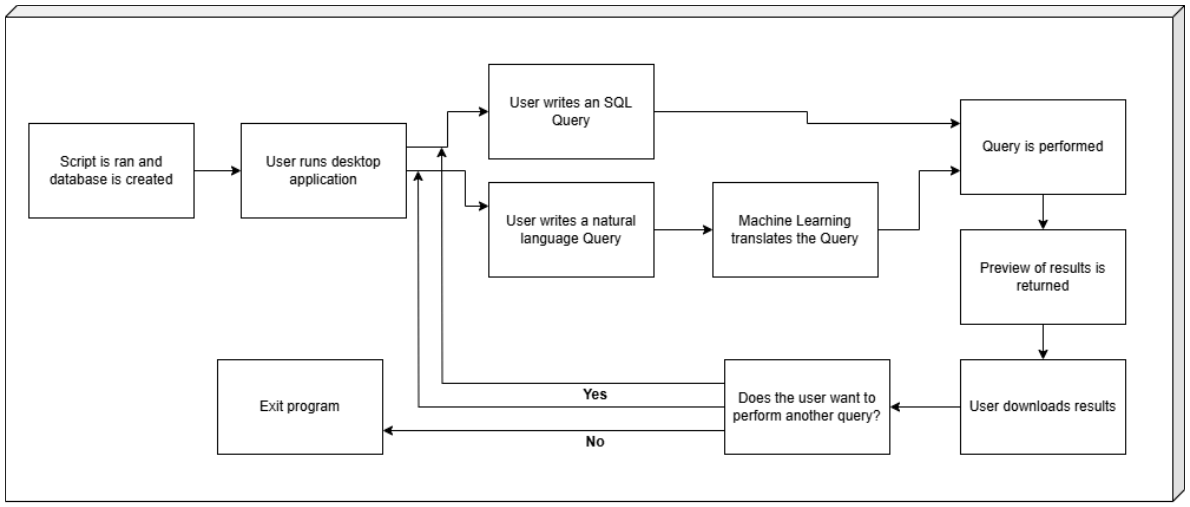

Script is ran and database is created
User runs desktop application
User writes an SQL Query
User writes a natural language Query
Machine Learning translates the Query
Query is performed
Preview of results is returned
User downloads results
Does the user want to perform another query?
Yes
No
Exit program

Figure 4: Task flow

4.3.4 Areas of Concern and Development

Going into this project, we determined our requirements and user needs. At its core, our requirements are to build a database that stores the data generated by POSYDON simulations and to allow users to easily retrieve data from this database. Beyond this, we noted that non-technical users would make up a large portion of our user base. As a result, we created several additional requirements to support these users. These users need to be able to set up the database without necessarily knowing how to maintain a database, and they need to have the option of querying the database without necessarily having a detailed knowledge of SQL and its syntax.

Our project fulfills these requirements through each of its major components. To solve our initial requirements, we chose to store the data in a PostgreSQL database. As a result of its relational nature, SQL is easy to develop queries for, ensuring the database consistently retrieves all desired data when queried. This is handled even further through our frontend user interface. Through this, users are able to enter queries without interfacing with the database directly. Similarly, this user interface supports our requirement of allowing non-technical users to query the database. By integrating our user interface with OpenAI via a textbox for natural language, users can find relevant information from the database without writing SQL queries. Furthermore, these users can initially set up the database through our database generation script. This ensures users have no need to understand relational databases to set this up.

Given this design, our primary area of concern is our reliance on an LLM. Though GPT is improving over time, it does not always follow prompts, and it is possible for users to trick it. As a result, we opted to develop software for our backend application that verifies that queries performed are properly structured so that they adhere to our database schema. This will be done through a two-stage process in which our backend application performs initial checks to confirm the validity of the query. If the application finds no issues with the query, the backend will utilize the PostgreSQL database management system to confirm the query is valid. We believe that this will substantially mitigate the risk to the project posed by the use of an LLM.

4.4 Technology Considerations

This section describes the distinct technologies used in our design, including their strengths, weaknesses, trade-offs, and any alternative solutions considered.

4.4.1 Windows

Windows is a widely used, industry-standard operating system requiring significant computational power to run effectively on modern hardware. Given its performance capabilities, a personal computer with an up-to-date version of Windows provides adequate processing power to run our program, initialize the database, and execute queries efficiently. Windows also offers compatibility with all the frameworks in our project and simplifies GUI design. However, a major drawback of using Windows is its lack of cross-compatibility with other operating systems, such as Linux and macOS, which some users may prefer.

4.4.2 Python

Python is a high-level programming and scripting language known for its flexibility and ease of use. In this project, Python serves as the main backend language, providing frameworks that facilitate connections between the user interface, backend database, and OpenAI's API. Python's cross-platform compatibility allows it to be installed on virtually any modern operating system. However, Python's high-level nature results in increased overhead, particularly when compared to lower-level languages like C, C++, or Java, potentially impacting performance in high-demand use cases.

4.4.3 PostgreSQL

PostgreSQL is a powerful, open-source relational database management system (RDBMS) chosen for its secure and reliable data storage, robust support for complex queries, and ability to handle large datasets—ideal for research environments. Its ACID compliance ensures data integrity and flexibility. However, PostgreSQL's complexity requires careful tuning for optimal performance, particularly with intricate datasets like those from the POSYDON Project. This can increase the need for regular maintenance and updates to ensure proper storage and organization. These trade-offs between advanced functionality and the need for diligent optimization are key considerations in our design.

4.4.4 OpenAI (ChatGPT)

OpenAI's ChatGPT provides extensive support for natural language processing (NLP), allowing our system to assist users with minimal SQL knowledge to interact seamlessly with the database. This integration enhances the user experience by providing an accessible, AI-driven interface for those who need it or those who want ease of use. ChatGPT's API enables streamlined responses by associating queries with a unique API key. This key can be shared among people, or individuals may purchase one for their own personal needs. By using this unique API key, individuals are able to offload computational load to OpenAI's servers, reducing local system overhead. Compared to other large language models, ChatGPT stands out for its ease of setup, low computational overhead, and outstanding customer support.

4.4.5 React

React is a JavaScript (JSX) library developed by Meta (Facebook) to facilitate the development of single-page applications (SPAs) with streamlined user interfaces. React's component-based architecture enables modularity and code reusability, reducing the need for redundant code and allowing dynamic page rendering with minimal updates. Additionally, React's extensive library ecosystem offers pre-built components for customized user interface design. However, React's reliance on JSX presents a learning curve for our team, as it is a new technology for most members. Despite this challenge, React's modular approach and rich feature set make it a promising choice for developing a responsive, maintainable interface.

4.5 Design Analysis

Our final design has the potential to work well with further testing. By utilizing Figma, we were able to determine an initial design we can iterate on as we build the project. As our project involves large data sets, we are limited by the amount of data we can test due to hardware limitations. As we develop our project, we anticipate the implementation of the Figma design to change. We will also likely encounter some unforeseen issues when working with large datasets, which will be addressed as our development progresses.