Near Earth Objects: Mapping the Future of Astronomical Encounters

Nakul Mody, Matthew Thomas, Jacky Lin

May 2, 2025

1 Introduction

Near Earth Objects (NEOs), such as asteroids, comets, and meteoroids, have become increasingly popular in scientific research and public attention in recent decades. While most NEOs are harmless, larger objects pose significant threats to the safety of our planet. Impacts of these larger objects can cause immediate destruction following their impacts. For example, their impacts can launch dust and debris into the atmosphere, leading to climate disruptions, potentially triggering critical environmental changes in the future. Impacts in the ocean can also cause tsunamis, bringing immediate dangers to coastal areas. Given the potential for catastrophic damage, it is important to understand the nature and frequency of future NEO events to better prepare for encounters. Through this project, we aim to present an improved understanding of future NEOs in hopes of supporting researchers contributing to advancing scientific research and knowledge in space sciences.

2 NEO Data

The Near Earth Object data set used for this project comes from the Center for Near Earth Object Studies (CNEOS). CNEOS determines the times and distances of all NEOs after 1900 A.D. and before 2200 A.D. In this project, however, we are only focusing on data based on future events.

Our focus is specifically on NEOs that will come within a distance of 0.5 AU (astronomical units) or less, which represents a threshold for potentially hazardous objects. Currently, the dataset comprises approximately 16,400 entries, each corresponding to a unique NEO. Each entry includes several critical fields that provide insights into the characteristics of these objects.

Key fields in the dataset used in this project include the Close-Approach Date (with time uncertainty), the closest approach distance (in astronomical units), and the minimum close approach distance, indicating how near the NEO will pass by Earth. Moreover, crucial information such as the speed of the NEO relative to the mass of the Earth at the time of its closest approach, as well as the estimated size range of the NEO, are provided, both of which are essential for evaluating the severity of any potential impacts.

3 Key Technologies

This project utilizes several key technologies to ensure that the application is efficient, portable, and capable of running successfully on any machine. At its core, the project is built around a Flask API, which handles user requests and returns information accordingly. Flask is a lightweight web application framework that enables developers to efficiently create APIs from the ground up. In this project, Flask serves as the backbone of the system, allowing users to interact with the server through defined routes and retrieve Near Earth Object (NEO) data.

To ensure consistency and portability across different environments, the application is containerized using Docker. Docker is a platform that packages applications and their dependencies into containers, guaranteeing that the project runs the same way regardless of the system. By using Docker, the project becomes easier to deploy, reproduce, and maintain.

Redis is used to maintain data persistence and manage job queues by serving as an in-memory key-value database. Its use significantly improves the efficiency and simplicity of accessing and managing data. Even when the API is not running, Redis ensures that data is not lost by persisting it to disk. Additionally, Redis enables multiple processes to access and operate on the same dataset. Job management is further facilitated through the hotqueue module, which utilizes Redis databases to create and monitor job analysis. This project utilizes four different Redis databases; raw NEO data, job queue, job information, and job output.

Finally, Kubernetes is used to orchestrate and manage these containers at scale. Kubernetes is a platform for deploying, scaling, and managing containerized applications automatically. It ensures that the correct number of containers are running, distributes them across machines efficiently, and helps recover from failures. Together, Flask, Redis, Docker, and Kubernetes create a scalable and portable application.

4 Endpoints

The Flask application includes multiple routes that allow users to request specific NEO data. The overall application design ensures that users can easily access and interpret the data. By structuring the API to allow customized queries, the application also serves as a valuable tool for data processing and analysis. The API provides several endpoints for retrieving and analyzing NEO data. The summary below outlines the available routes and their functions.

Table 1: API Endpoints for NEO Database

Endpoint	**Method**	**Description**
`/data`	GET, POST	This is both a GET and a POST endpoint. The user may make a GET request to retrieve the data in the Redis database. The user may make a POST request to mount the data on the Redis database from the local CSV file. The database is stored locally in the `data` folder.
`/data/<year>`	GET	The user may curl this endpoint, including a year as a parameter and would receive all NEOs spotted during that year.
`/date/date`	GET	The user may curl this endpoint and retrieve all the dates and times for all the NEOs.
`/data/distance`	GET	The user may curl this endpoint using the `min` and `max` parameters to find all the NEOs that are between these distances in astronomical units.
`/data/velocity_query`	GET	The user can curl this endpoint with the `min` and `max` parameters to find all NEOs with velocities between the specified values. The units for this query are kilometers per second.
`/data/<max_diameter>`	GET	The user can curl this endpoint given they provide a max diameter value (float) and receive all the NEOs with max diameter less than the input.

(Continued on next page)

(Continued from previous page)

Endpoint	**Method**	**Description**
`/data/biggest_neos/<count>`	GET	The user can curl this endpoint given they provide an integer and receive the biggest "x" number of NEOs where "x" is the input. This is based on the H magnitude scale.
`/now/<count>`	GET	The user can curl this endpoint given they provide an integer and receive the "x" number of NEOs closest to the current time where "x" is the input.
`/jobs`	GET, POST	If the user curls the GET version, the endpoint will return all the jobs in the queue along with their status. If the user curls the POST request with the correct parameters, a job will be added to the hotqueue in the Redis database.
`/jobs/<jobid>`	GET	The user can curl this endpoint given they provide a valid job ID and receive the status of the specific job.

5 Usage

Users should navigate to the NASA NEO website found in the README file. The website contains a data table containing all past and future NEO's that users can filter to their liking. For our project, we request that users filter the data to only include NEO's less than 0.05 au and approaching in the future before downloading it as a csv.

Users can run our application by cloning the Github project repository at JYL2027/Near-Earth-Objects-API.git and then utilizing it as their working directory. The repository is well organized, with a /src directory containing all files that define the Flask API route, worker, and job functionalities. Users should rename the raw data csv to 'neo.csv' and move it into the empty /data directory. These steps are important because the Docker image building process requires that the csv be located and named correctly. Since the API is containerized, users must also ensure that Docker is installed on their system before proceeding.

To begin operations, users must first pull our public image from Docker Hub by running the command: `docker pull mjt2005\neo_api:1.0`.

To build and run the container locally, users can type the command `docker compose build mjt2005\neo_api:1.0`. followed by `docker compose up -d` . The -d flag allows the container to be run in the background, freeing the terminal for curl routes. These commands spin together the Flask, Redis, and worker services. To deploy this application on a Kubernetes cluster, users should first ensure that they have access to Kubernetes. This project was originally created and deployed a cluster provided by TACC for use in the COE332 class. If users have access, they can navigate to the files located in the /kubernetes folder and change the <username> part of each file to their Dockerhub username. They must then run the command kubectl apply -f <file_name> for every file in the directory.

If the user does not have access to Kubernetes, they can still access this project through the public internet since the API was deployed on our own Kubernetes cluster. Users can curl routes in the browser using the hostname "neo-analysis.coe332.tacc.cloud". POST requests should be done from a terminal, however all GET requests can be done in a browser. Output will be sent to the webpage and can be viewed.

```
curl -X POST neo-analysis.coe332.tacc.cloud/jobs -d '{"start_date
   ":"2027-Jun-12","end_date":"2035-Jan-30","kind":"1"}' -H "
   Content-Type:application/json"
```

The above command shows an example job 1 POST

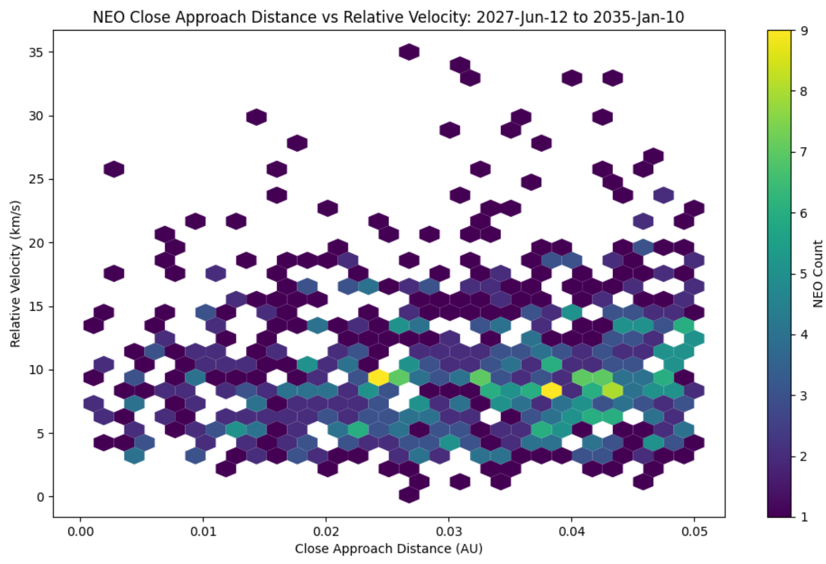

Figure 1: Example plot generated from a submitted job

5.1 Jobs

Our API is equipped with the functionality to support two different jobs. A job is a functionality that is handled asynchronously based on its departure from the queue. The workflow is as described:

1. A user submits a POST request to a job.
2. The program adds the job to the queue.
3. Using the First In, First Off principle, submitted jobs are removed from the queue and handled asynchronously by the worker.
4. The results of the job are saved in byte form to the results database.

Jobs are meant for longer, computationally expensive tasks that could not normally be handled by an HTTP route. In this project, the job the user wishes to POST is specified with a start date, end date, and kind parameter. In both jobs, the start and end date parameters must be strings in the form "YYYY-MMM-DD". Users can save the results of a job to their local computer through the /results<jobid> route. The /jobs and /jobs<job_id> routes are useful in managing the several jobs that can be running concurrently.

5.1.1 Job 1

Job 1 creates a hexbin graph portraying the density of relative velocities and the near approach distances of NEOs within that date range. This job will accept any range of dates, and is nicely color coded based on density of NEOs.

5.1.2 Job 2

Job 2 creates a scatter plot giving an overview of the NEOs approaching in a given month. The x-axis is the day of the month and the y-axis is the relative velocity. The color and size of the dots correlate to its Rarity (0-3), and magnitude of the NEO, respectively. This job will only accept a start date and end date that are in the same month, as this plot is intended to display at most, a month's worth of NEO data.

6 Ethical and Professional Responsibilities

One of the main concerns in engineering applications is the consistency and validity of the information being presented. If a product is not accurately described or documented, it can lead to misunderstandings or mistakes that could have significant consequences. Engineers and developers must consider the impact their work has on users, society, and the field of technology. By ensuring data accuracy and using data responsibly, engineering applications can continue to grow without the risks associated with small errors. The NEO API adheres to this standard by ensuring transparency in data usage and reliability in its implementation. By sourcing trajectory data directly from NASA and maintaining clear documentation, this project helps users understand where the data comes from, preventing misinformation. Additionally, the use of error handling in the scripts ensures that incorrect inputs are managed, supporting the ethical responsibility of providing accurate and reliable information.

A second key aspect of maintaining ethical responsibility and professional practice is being transparent and open about the limitations of the data. The developers of this project recognize that the data provided contains inherent bias and potential inaccuracies. Since the dataset involves predictions about future events, it cannot be treated with complete certainty and should be analyzed with caution. Additionally, all scientific data carries some level of bias—whether from the methods of collection or the interpretation of results. For example, extreme data points may be disregarded as "errors," when in fact they could represent valid but rare phenomena. Being open about these uncertainties allows users to make more informed and ethical decisions based on the data.

Another key aspect of professional responsibility in engineering is ensuring that a product is reliable, functions as intended, and minimizes the potential for user error or system failure. Engineers have an ethical obligation to deliver applications that meet user needs without requiring additional troubleshooting or technical investigation. By doing so, they help maintain public trust in engineering solutions and support the integrity of the tools used by other professionals. This project reflects that commitment through comprehensive testing, robust error handling, and the use of defensive programming. Defensive programming involves anticipating edge cases, validating inputs, and building safeguards to prevent misuse or unexpected behavior. These practices ensure that the software behaves predictably, even under unusual conditions, thereby reducing the likelihood of failure. Together with well-structured pytests, this approach reinforces ethical standards by prioritizing user safety, product dependability, and long-term sustainability.

7 Design Principles

This project incorporates key software design principles such as modularity, abstraction, generalization, and portability. The project contains modularity at a low level in the sense of all API endpoints are contained within the NEO_API file and all job related functions in the job file. On a higher level, each file tackle a different "job" as the API file handles the endpoints, the jobs file handles the breakdown of jobs on the queue, and the worker handles processing and doing the job. Together they work together to create one cohesive system.

The idea of abstraction is incorporated in the project on a broader level in the sense of the user hitting endpoints. For example, the user may post a job, but when doing this, they only have to worry about giving the correct parameters. Internally, creating the job is not the end of the process as it has to be stored in the redis database on the hotqueue. In a more specific way, the coders use libraries such as "json", "requests", and more to help create the project. While the developers may not be familiar with the intricate details of how each function is implemented, they can effectively utilize these libraries because they understand their purpose and functionality.

The principle of portability is implemented in this project through the use of Docker. By creating a Dockerfile, the entire system is encapsulated in its own environment, meaning that all necessary dependencies are contained within the Docker container itself. This allows the program to run consistently on any system without requiring external installations or configurations, making it easy for anyone to execute the project regardless of their environment.

The software principle of generalization is demonstrated in this project through the organization of the data. Since all the data provided by NASA included some form of uncertainty, it made querying the values more complicated. Dates, which were often used for queries, had relatively low uncertainty, so the developers decided to generalize the data by omitting the uncertainty values. This decision simplified the querying process, as developers could focus on the dates themselves without worrying about potential variations due to uncertainty, thus creating a more consistent and streamlined data structure. By generalizing the data in this way, the system became more flexible and efficient in handling a variety of queries."

8 Citations

[1] Center for NEO Studies. (2019). Nasa.gov. `https://cneos.jpl.nasa.gov/`