In Your Orbit

Tech Feasibility

Cloud-Based Planetary Ephemerides

United States Geological Survey

Mentor: Scott Larocca

Austin Carlile
Noah Schwartz
Minuka Trikawalagoda
Nicholas Gonzalez

The purpose of this tech feasibility document is to introduce the project, detail the technical challenges, analyze the options and explain the decisions used to solve them

10/18/2024

Table of Contents

1. Introduction

NASA's missions generate large amounts of planetary data, all of which is critical to mapping, photogrammetry, and navigation in planetary science. Instrument Support Data (ISD) is heavily relied on for accurate data collection since it provides precise measurements and positioning information from the SPICE database. However, accessing the ISD currently involves quite large downloads–up to multiple terabytes in size–which poses a significant challenge for scientists, especially those working in environments where high-performance computing is a must. This approach requires extensive storage capacity and time, diverting resources from the critical tasks of analysis and mission planning.

Existing solutions to access ISD are quite inefficient and often require special tools and massive downloads to local machines. While NASA does provide access to SPICE kernels, there is no streamlined and efficient way for scientists to retrieve only the essential data for their research without having a large overhead of entire datasets. As a result, these scientists often face roadblocks for tasks that are mission-critical which significantly hinder the processing of the planetary images.

Our proposed solution to these challenges is a web-based RESTful service that will facilitate ISD generation and retrieval in a much less time consuming manner. The service will use Python to process the image parameters, generate the ISD, and deliver the optimized data format to the users. One of the key features will be a caching system that will significantly reduce ISD generation time by storing previously retrieved data. In addition to the caching server, we will utilize data compression to minimize transmission footprint, and deploy the whole system on AWS in order to leverage cloud scalability. By addressing these challenges, the service will significantly reduce the data access time, streamline the data workflows, and eliminate the need for so much local storage being used, ultimately optimizing NASA's mission objectives.

Given that the data we are working with is quite complex, the following sections will attempt to break down our solution into some smaller parts that we dive deeper into. This document outlines our tech feasibility analysis, starting with an overview of our challenges, followed by an examination of each system we will be using as well as alternatives.

2. Technological Challenges

There are several technological challenges that are presented by this project that need to be addressed to ensure success. First of all, we will need to create a RESTful web service which will serve as the main pathway for our application. Amazon AWS servers will be the host of this web service which is crucial for leveraging cloud capabilities and to ensure high availability.

In addition to hosting, we will require a caching server that supports cache validation and memoization to optimize data retrieval and enhance performance. To efficiently transmit data, we will also need to develop a compressed JSON data format. Furthermore, establishing a robust testing mechanism for the web service is crucial to ensure its reliability and functionality. Finally, scaling the web service to handle up to two hundred thousand simultaneous requests will be vital for accommodating a large user base and maintaining a smooth user experience. By tackling these challenges, we can build a scalable and efficient solution.

3. Technology Analysis

3.1 RESTful Web Service

3.1.1 Introduction

Representational State Transfer (REST) is a widely adopted architectural style that supports high-performance and reliable communication at scale. RESTful web services provide a straightforward framework that can support large numbers of simultaneous requests with minimal overhead, ideal for cloud-based applications.

3.1.2 Desired Characteristics

The core principles of REST align well with our needs for scalability and flexibility.

- **Uniform Interface:** REST's uniform interface allows consistent interaction regardless of underlying processes.
- **Statelessness:** Each client request is treated independently, reducing the need to store session state and enabling horizontal scaling.
- **Layered System:** Clients connect through intermediaries, allowing for load balancing and caching that optimize performance.
- **Cacheability:** REST enables data caching for repeated requests, reducing server load.
- **Code on Demand:** REST can transfer executable code to clients, enhancing client functionality and enabling dynamic behavior.

These principles make REST particularly suitable for this project by supporting **scalability, flexibility,** and **independence** in the underlying technology.

3.1.3 Alternatives

In selecting a web service architecture, we considered the REST style alongside other alternatives. After deciding on REST, we also evaluated two frameworks for implementing it, **Flask** and **FastAPI**, based on performance, scalability, and ease of use. The following are key options considered:

- GraphQL
    - This query language allows clients to request specific data fields, optimizing the query process by reducing over-fetching. However, its complexity in caching and overhead in simple services make it less suited for this straightforward, scalable API.
- gRPC
    - A high-performance RPC framework utilizing HTTP/2 and protocol buffers, suitable for low-latency microservices and streaming applications. While fast, gRPC's complexity, lack of browser support, and its reliance on client libraries limit its applicability compared to REST for this project.
- Flask
    - A lightweight and flexible Python framework for RESTful services, Flask is widely supported and has a large developer community. While straightforward, Flask has limited scalability out of the box and lacks built-in asynchronous support, making it less performant under high concurrency.
- FastAPI
    - A modern Python framework designed for RESTful services, FastAPI supports asynchronous programming and includes automatic OpenAPI documentation. Its performance and scalability make it ideal for handling high traffic and dynamic requests, aligning well with this project's needs.

3.1.4 Analysis

The alternatives above were considered based on project requirements for simplicity, scalability, and ease of integration. Flask and FastAPI were selected for further comparison, as they align most closely with our technical goals. While both **Flask** and **FastAPI** can support the deployment of our RESTful API, here we analyze how optimal each framework will be for our solution:

Flask:

- Moderate performance
- Moderate scalability
- No native asynchronous support
- Basic documentation generation
- High ease of use
- Established community support
- Suitable for simple or small-scale applications

FastAPI:

- High performance
- High scalability
- Native asynchronous support
- Automatic documentation generation
- High ease of use
- Growing community support
- Ideal for high-load, scalable applications with concurrent request handling needs

3.1.5 Chosen Approach

The frameworks were evaluated on **versatility**, **scalability**, **customizability**, and **complexity** to determine the best fit for implementing the RESTful API.

	Versatility	Scalability	Customizability	Complexity
Flask	Medium	Medium	High	Low
FastAPI	High	High	High	Medium

- **Flask**: Flask offers high customizability and ease of use, but its moderate scalability and limited versatility (especially in handling asynchronous tasks) make it less suited for high-performance scenarios.
- **FastAPI**: FastAPI excels in scalability and versatility with built-in asynchronous support and automatic documentation. While slightly more complex, it is optimized for high-performance APIs, making it the preferred choice.

Each category was ranked on a scale of High to low and a color was assigned to each ranking indicating the desirability (Green is desirable, Yellow is neutral and Red is bad). Based on the rankings of the four categories in the table above, **FastAPI** is selected for its ability to handle high concurrency, versatile features, and scalability, which align well with the project's requirements.

3.1.6 Proving Feasibility

To prove the feasibility of using FastAPI for the RESTful API, we will develop a small prototype of the web service. This prototype will be fully deployed and tested on AWS, incorporating key features like asynchronous requests, caching with Redis, and integration with the ALE library.

By testing this prototype in a cloud environment, we can validate FastAPI's performance, scalability, and ability to handle concurrent requests effectively.

3.2 Amazon AWS Server

3.2.1 Introduction

In order to allow users to access and use our web service, it will need to be hosted on an external server. Hosting the web service on the cloud will allow us to make it accessible and allow us to easily modify and scale it after release.

3.2.2 Desired Characteristics

- **Easily Scalable:** Our web service hosting needs to be easy to scale for thousands of simultaneous requests, so we will need to host it on a robust and flexible platform.
- **Easy to maintain:** We need to be able to easily update our web service and make changes as needed, so the hosting platform must be flexible.
- **Zero cost:** We need to be able to host our web service for free.

3.2.3 Alternatives

Using Amazon AWS services is a specified requirement by our client, so all alternatives are various Amazon AWS services. These are:

- EC2
    - General purpose instance hosting developed by Amazon in 2006.
- Elastic Beanstalk
    - Web Application host and manager developed by Amazon in 2011.
- ECS
    - Container orchestration manager developed by Amazon in 2006.

3.2.4 Analysis

While all three services are able to support the deployment of our web service, here we analyze how optimal they will be for our solution:

- **Elastic Compute Cloud (EC2)**:
    - Low versatility
    - Low customizability
    - Low complexity
    - Needs external service to scale
    - General Purpose
    - Simple to use
    - Non-Scalable by itself

- **Elastic Beanstalk**:
    - Medium versatility
    - Medium customizability
    - Medium complexity
    - Scales automatically
    - Can use EC2 instance or other server type
    - Dynamically configures resources needed

- **Elastic Container Service (ECS)**:
    - High versatility
    - High customizability
    - High complexity
    - Can use EC2 and Elastic Beanstalk
    - Can integrate many services to handle all aspects of application deployment
    - Can use multiple different services for scaling
    - Handles everything related to server and application

3.2.5 Chosen Approach

The two main aspects that differentiated the three services was degree of versatility and degree of scalability as the most important distinctions with the level of customizability and complexity as slightly less important but still worth considering. The ideal service would have high degrees of versatility, scalability and customizability, while having a low degree of complexity.

The following is a table of their ratings:

	Versatility	Scalability	Customizability	Complexity
EC2	Low	Low	Low	Low
Elastic Beanstalk	Medium	High	Medium	Medium
ECS	High	High	High	High

The table shown above ranks the three Amazon AWS services by their versatility, scalability, customizability and complexity. The four categories appear in order of importance from the greatest importance (Versatility) to the least importance (Complexity). Each category was ranked on a scale of High to low and a color was assigned to each ranking indicating the desirability (Green is desirable, Yellow is neutral and Red is bad). Based on the rankings of the four categories in the table above, ECS was chosen as our solution to the Amazon AWS service problem. Although it has a high degree of flexibility, its extremely high versatility, scalability and customizability, along with its ability to deploy and utilize both EC2 and Elastic Beanstalk if needed, make it a suitable choice to implement our web service.

3.2.6 Proving Feasibility

To prove the feasibility of the Amazon ECS (Elastic Container Service), we will be producing a small prototype of the web service and completely deploy and test it using the various services contained inside of ECS.

3.3 Caching Server

3.3.1 Introduction

In order to increase the efficiency of our web service, we will need to create a caching server that will hold and return previously generated ISD images. This will greatly reduce the computing power and time needed to return data to the user.

3.3.2 Desired Characteristics

- **Easily Scalable:** The caching server for our web service must be able to handle hundreds of thousands of simultaneous requests
- **Fast:** The caching server must be fast and able to return results to the user quickly
- **Low Complexity:** We need to be able to easily update and query our cache, so the caching server must have a low complexity.

3.3.3 Alternatives

Using Amazon AWS services is a specified requirement by our client, so all alternatives are various Amazon AWS services. These are:

- Elasticache
    - In-memory data store and cache service developed by Amazon in 2011.
- DynamoDB
    - NoSQL database service developed by Amazon in 2012.
- MemoryDB
    - In-memory database service developed by Amazon in 2021.

3.3.4 Analysis

While all three services are able to support the deployment of our web service, here we analyze how optimal they will be for our solution:

- **Elasticache**:
    - High scalability
    - High speed
    - Low complexity
    - Automatically scales

- **DynamoDB**:
    - High scalability
    - Medium speed
    - High complexity
    - Automatically scales

- **MemoryDB**:
    - High scalability
    - Medium speed
    - High complexity
    - Manually scales

3.3.5 Chosen Approach

The main aspects that differentiated the three different services were degrees of scalability, speed and complexity. Another important aspect of each service was the style of scaling that they employed, either automatic or manual. The ideal service would have a high degree of scalability and speed, while having a low degree of complexity and an ability to automatically scale.
The following is a table of their ratings:

	Scalability	Speed	Complexity	Auto-Scaling
Elasticache	High	High	Low	Yes
DynamoDB	High	Medium	High	Yes
MemoryDB	High	Medium	High	No

The table shown above ranks the three Amazon AWS services by their Scalability, Speed, Complexity and ability to Auto-Scale. Each category was ranked on a scale of High to low and a color was assigned to each ranking indicating the desirability (Green is desirable, Yellow is neutral and Red is bad). Based on the rankings of the four categories in the table above, Elasticache was chosen for the solution to our caching server problem. Elasticache fit all requirements for our solution and was ideal in each category.

3.3.6 Proving Feasibility

To prove the feasibility of using Amazon Elasticache, we will be deploying a small caching server of ISD test data and will test the scalability and response time of the caching server.

3.4 Compressed JSON format

3.4.1 Introduction

The use of compressed JSON is essential to handle large quantities of Image Support Data (ISD) efficiently. Given the constraints of handling planetary imagery data for NASA missions, minimizing the data footprint while retaining essential metadata is crucial. The compression process aims to make data transmission over the web faster, reduce storage requirements, and support scalable requests.

3.4.2 Desired Characteristics

- **Compact Representation**: Uses fewer characters and employs techniques like removing unnecessary whitespace and key reduction.
- **Data Integrity**: Maintains the structure of ISD files without losing critical information required for sensor models.
- **Interoperability**: Can be unpacked back into JSON formats supported by the ALE (Abstract Layer for Ephemerides) library.
- **Ease of Use**: Reduces the data size without requiring end-users to understand the compression algorithms.
- **Scalability**: Supports thousands of simultaneous requests, critical for scientific workloads.

3.4.3 Alternatives

- Binary Formats (e.g., CBOR, BSON)
    - Faster serialization and deserialization than JSON; however, harder for humans to interpret.
- MessagePack
    - Compact and efficient binary-based alternative, but requires more specialized libraries to use.
- Gzip Compression
    - Can be applied directly to JSON files, but adds overhead during decompression.
- Protocol Buffers or Avro
    - High performance but requires schemas and adds complexity in data encoding/decoding.

3.4.4 Analysis

- **Performance Impact**
    - Compressing JSON reduces payload sizes, resulting in faster transmission and lower AWS bandwidth costs.
- **Data Integrity**
    - The chosen approach must ensure complete and accurate reconstruction of ISD files after decompression.
- **Implementation Complexity**
    - Adding compression libraries like zlib or switching to MessagePack will increase complexity but offer higher performance.
- **Caching Benefits**
    - Compressed data can be cached to reduce the load on the backend, improving response times.
- **Scalability Requirements**
    - Given the high frequency of ISD generation, the format should support batch processing without significant performance degradation.

3.4.5 Chosen Approach

The main aspects that differentiated the four techniques were their scalability, speed, complexity, and auto-scaling capability. The ideal approach would offer high scalability and speed, with a low level of complexity and support for automatic scaling to handle varying workloads efficiently.

	Scalability	Speed	Complexity	Auto-Scaling
JSON Minification	High	High	Low	Yes
GZip on Transmission	High	Medium	Medium	Yes
Message pack for Storage	High	High	High	No
Memoization Strategy	High	High	Medium	Yes

The table above ranks the four approaches by their scalability, speed, complexity, and auto-scaling capability. Each category was ranked on a scale of High to Low, with colors indicating desirability (Green is desirable, Yellow is neutral, and Red is less desirable). Based on these rankings, a combination of JSON Minification, Gzip on Transmission, and Memoization Strategy was selected as the best solution. This approach ensures high performance with scalability and auto-scaling support while keeping complexity manageable.

3.4.6 Proving Feasibility

To prove the feasibility of using JSON minification, Gzip on transmission, and memoization, we will produce a small prototype of the web service and deploy it with these compression techniques. This prototype will be tested on a server using Redis for caching, allowing us to evaluate scalability, response time, and data handling efficiency under various load conditions.

3.5 Testing Mechanism

3.5.1 Introduction

Using Docker as the chosen testing mechanism for the RESTful web service designed to process planetary image labels from NASA's SPICE system. Docker offers a consistent and isolated environment for mimicking production, and can be highly optimized for AWS infrastructure which is essential for testing a service that processes large datasets in a short amount of time.

3.5.2 Characteristics

- **Portability:** Docker containers ensure the service runs consistently in different environments, which is essential for high-performance computing needs across different environments.

- **Isolation:** Each service including the web service, caching system, and ISD data processing components can be tested in isolated containers, preventing dependency conflicts.
- **Reproducibility:** Docker offers the possibility to reproduce the exact environment used in testing for debugging.
- **Data Management:** Docker containers can simulate scenarios of handling large datasets up to a terabyte. This ensures the data compression and caching systems can work as expected.
- **Integration Testing:** Docker provides an environment to test the interaction between the caching system and the other processes, ensuring performance optimization on AWS.

3.5.3 Alternatives

- Virtual Machines
    - Virtual Machines provide isolated environments but are resource-heavy and slower, especially when testing large datasets.
- Native Testing
    - Running tests on the development machines could work but lacks consistency, especially across multiple machines.
- Kubernetes
    - Kubernetes offers container organization but is less documented than Docker, especially in pairability to AWS. This may lead to unnecessary complexity that will be easier to solve with Docker.

3.5.4 Analysis

- **Efficiency**
    - Docker's lightweight containers make it easier to test the web service's performance without the overhead of managing full VMs. This is crucial for testing the large datasets, improving efficiency.
- **Data Handling**
    - Containers can simulate environments where up to a terabyte of data is handled, ensuring that compression, caching, and AWS optimization are thoroughly tested.
- **Scalability Simulation**
    - Docker can simulate AWS scaling by running multiple containers to mimic high traffic, thus stress testing the system's response.

3.5.5 Chosen Approach

The main aspects that were required for a testing tool for this system were low overhead, scalability, consistency across machines, and being able to handle large data loads. The ideal service would be one that could handle large datasets with a low overhead, as well as be able to test the system for multiple machines, and scale as needed.

The following is a table of ratings for native testing, virtual machines, and Docker:

	Overhead	Scalability	Consistency	Data Processing
Native Testing	Medium	Low	Low	Low
Virtual Machines	High	Medium	Medium	High
Docker	Low	High	High	High

The table shown above ranks the three testing options by Overhead cost, scalability, consistency, and data processing potential. Each category was ranked on a scale of High to low and a color was assigned to each ranking indicating the desirability (Green is desirable, Yellow is neutral and Red is bad). Based on the rankings of the four categories in the table above, Docker was chosen as our testing service as it fits all requirements and is desirable for all four standards.

3.5.6 Proving Feasibility

To prove the feasibility of Docker as a testing service, we will be setting up containers representing the RESTful web service, the caching system, and all other services related to compression and ISD data. They will be able to handle large volumes of data, test the caching system, and simulate AWS scaling features.

3.6 Conclusion

The technology analysis thereby chose the following as the best toolset to implement a scalable and efficient planetary ephemerides service: RESTful web services, AWS ECS, ElastiCache, and JSON compression. These technologies have been chosen for their simplicity, good performance, and capability of supporting high-traffic cloud environments. REST will ensure reliable client-server communication, AWS will provide dynamic scaling, and ElastiCache will optimize response times by keeping frequently requested data in the cache. The application will ensure that with such synergistic working of these components, the proposed system handles big datasets with much efficiency, reduces computational overheads, and provides fast access to ISD data to scientists so that they can effectively support NASA's missions.

4. Technology Integration

In order to create a high-performance coherent system, all the individual components discussed must be integrated into a unified architecture. Our main challenge is ensuring that the RESTful web service, caching system, AWS infrastructure, and data handling mechanisms all work in unison seamlessly to meet the stringent performance and scalability requirements. Our system architecture will efficiently process large ISD files, optimize data retrieval, and scale dynamically based on current demand.

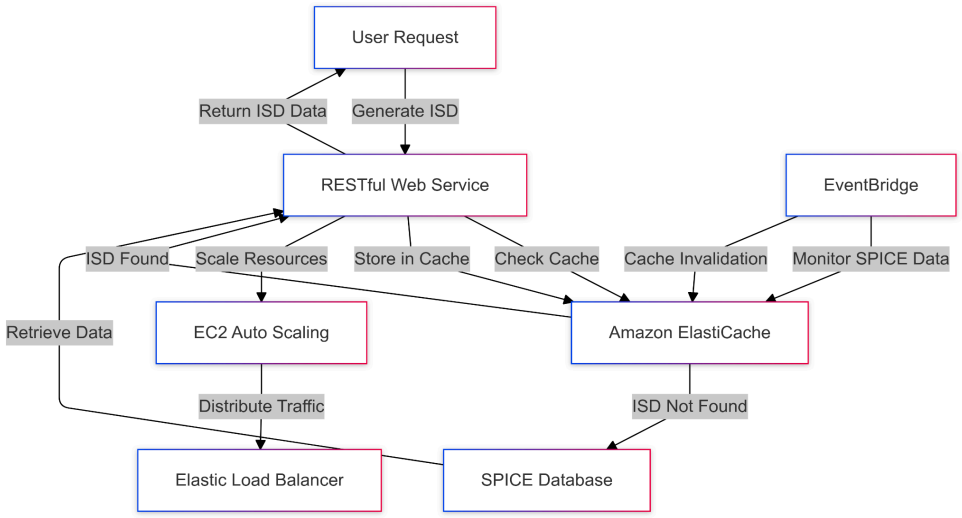
4.1 System Architecture Overview

At the core of this architecture is a RESTful web service, implemented using either Flask or FastAPI which will serve as the main entry point for various user requests. The service will be hosted utilizing Amazon EC2 instances, which will be managed using Amazon ECS for better scalability. The service will process planetary image labels and parameters from NASA's SPICE database, allowing users to generate ISD data without needing to download large datasets.

4.1.1 Elements of System Architecture

- Amazon Elasticache
- AWS EC2 Auto Scaling
- Elastic Load Balancing
- EventBridge
- Data Compression

4.2 System Diagram



4.3 System Components and Integration

4.3.1 Restful Web Service

The web service is the main entry point responsible for processing requests and interacting with the caching and database layers. It will handle requests for ISD data generation from the planetary image labels as well as compress ISD files for efficient transmission and storage. This service will scale automatically using AWS EC2 auto scaling to manage large requests and traffic efficiently.

4.3.2 Amazon EC2 and ECS

The web service will run on Amazon EC2 instances managed by ECS, which will offer scalability and flexibility. EC2 Auto Scaling will dynamically adjust resources based on traffic, which will ensure smooth operations during large requests, as well as performing cost efficiently in times of lesser traffic.

4.3.3 Elastic Load Balancing

Elastic Load Balancing will efficiently route network traffic and requests to the different EC2 instances managed by EC2 Auto Scaling, in order to maintain an equal load between all instances. This will greatly increase the scalability and efficiency of the web service.

4.3.4 ElastiCache

Amazon ElastiCache will store the frequently requested ISD files, which will reduce the time to serve repeat requests. Cached data will be compressed to minimize storage costs and response times. EventBridge will trigger cache invalidation when SPICE data updates occur, making sure that all fresh data is accounted for.

4.3.5 Data Flow Summarized

The system begins with a user request to generate ISD data, which will be initially handled by the web service. The service checks ElastiCache to see if the requested data is already available in its compressed form, and if it is, it is returned to the user immediately, reducing processing time. If not, the web service queries the SPICE database for the necessary data. The service then processes this data and stores the ISD in ElastiCache for future use. To maintain accuracy within our cache, Amazon EventBridge will monitor the SPICE database for updates, triggering cache invalidation and reprocessing when new data becomes available. Throughout this process, EC2 Auto Scaling will adjust the system’s resources dynamically, based on real-time traffic to ensure that the system can handle fluctuating traffic efficiently. Elastic Load Balancing will then route network traffic and requests to the different EC2 instances to make maximum use of each instance and stay at peak efficiency.

5. Conclusion

This document has presented the technological feasibility regarding the development of a cloud-based planetary ephemerides service for automating the generation and dissemination process of ISD. The proposed solution will integrate a number of key technologies that include but are not limited to RESTful web services, caching mechanisms, data compression techniques, and AWS infrastructure in order to overcome current generation processes of ISD.

Computation will also be minimized due to the use of a RESTful web service and the implementation of caching with Amazon ElastiCache. The use of JSON formats in compressed forms will reduce the data footprint, increasing the speed of transmission and efficiency in storage. Employing AWS services including ECS, Elastic Load Balancing, and Auto Scaling will keep the system ready for variable demand and seamless scalability.

The feasibility analysis indeed confirms that the selected technologies are able to meet project requirements within time and budget limits. The use of Docker containers for testing will ensure that the system is capable of emulating a production environment and verify that each component can be integrated. Careful design was to be realized, including the development of efficient caching strategies and AWS optimizations, to deliver fast, scalable, and reliable access to ISD for planetary scientists, enabling new research opportunities and reducing entry barriers for future scientists.

In other words, the above cloud-based planetary ephemerides service is going to offer a robust solution for the existing challenging problems of big data size and computation overhead. Once this is in place, the generation and access of ISD data by researchers will be much easier, thus engineering faster smoother planetary science research in order to assist NASA missions.