Pixxel Satellite Attitude Visualizer - System Design Document

1. Introduction

This document provides a comprehensive overview of the system architecture, software components, interface design, and underlying rationale for the Pixxel Satellite Attitude Visualizer. This tool was developed as part of the Mission Software Intern assignment to visualize satellite attitude data using quaternions, allowing users to compare actual and predicted values interactively.

2. Objectives

- Visualize 3D satellite attitude data derived from quaternion telemetry.
- Enable side-by-side comparison of actual vs. predicted attitude data across multiple files.
- Allow engineers to navigate time-based attitude data with playback controls and sliders.
- Provide export functionalities for further offline analysis.
- Deliver an intuitive, browser-accessible interface for technical and non-technical users.

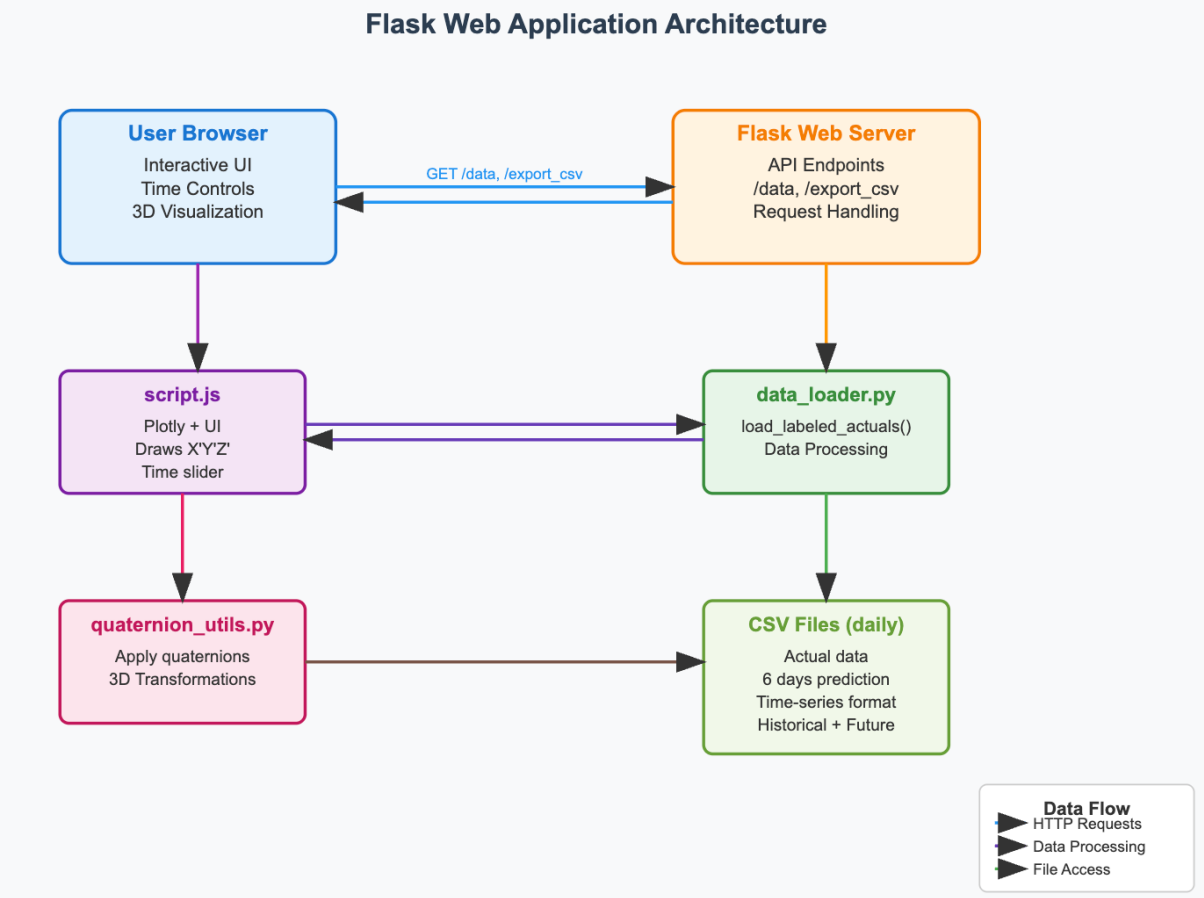
3. System Overview

The visualizer consists of a modular web-based architecture, split into data handling, computational processing, backend routing, and frontend rendering layers. It operates in batch mode, loading prediction and actual data from historical CSV files. The tool provides full interactivity via a client browser without requiring any installation beyond standard Python and browser environments.

Key Functionalities:

- Load and parse multiple CSV files, identifying and tagging actual vs. predicted rows.
- Apply quaternion rotations to unit basis vectors to generate X', Y', Z' orientation in 3D.
- Render a rotatable cube representing satellite attitude over time.
- Animate the rotation through a time slider with playback and speed controls.
- Display a timeline bar identifying file origin and type of each data point.
- Export visualized data (actual/predicted) and plots.

4. Architecture Diagram



The architecture follows a layered and modular design:

- **Prediction Model (Optional)**: Placeholder for future integration of on-demand prediction generation.
- **Data Loader**: Scans and parses input files, enforces format validity, tags rows.
- **Backend (Flask)**: Serves routes (`/`, `/data`, `/export_csv`) and provides structured data to the frontend.
- **Frontend (HTML/JS/Plotly)**: Renders the 3D visualization, time controls, and user interactions.

5. Component Breakdown

5.1 Flask Backend (`app.py`)

- **Routes**:
  - `/`: Serves main HTML interface.
  - `/data`: Sends stitched JSON object containing rotated vectors, timestamps, and metadata.
  - `/export_csv`: Triggers CSV export; supports filtering for "actual" or "predicted" rows.
- **Justification**: Chosen for simplicity, maintainability, and Python-native integration with `pandas` and `scipy`. Ideal for prototyping and small-scale deployment.

5.2 Data Loader (`data_loader.py`)

- **Functions**:
  - `load_labeled_actuals_and_predictions()`:
    - Loads CSV files from the `data/` folder.
    - Converts timestamps, sorts data.
    - Tags rows based on last-day cutoff logic (actual = final day; rest = predicted).
    - Supports downsampling for performance beyond 100,000 rows.
- **Design Rationale**:
  - Stateless design — no caching or in-memory persistence.
  - Allows flexibility in file formats as long as they contain `Q0`–`Q3` and `Timestamp` columns.

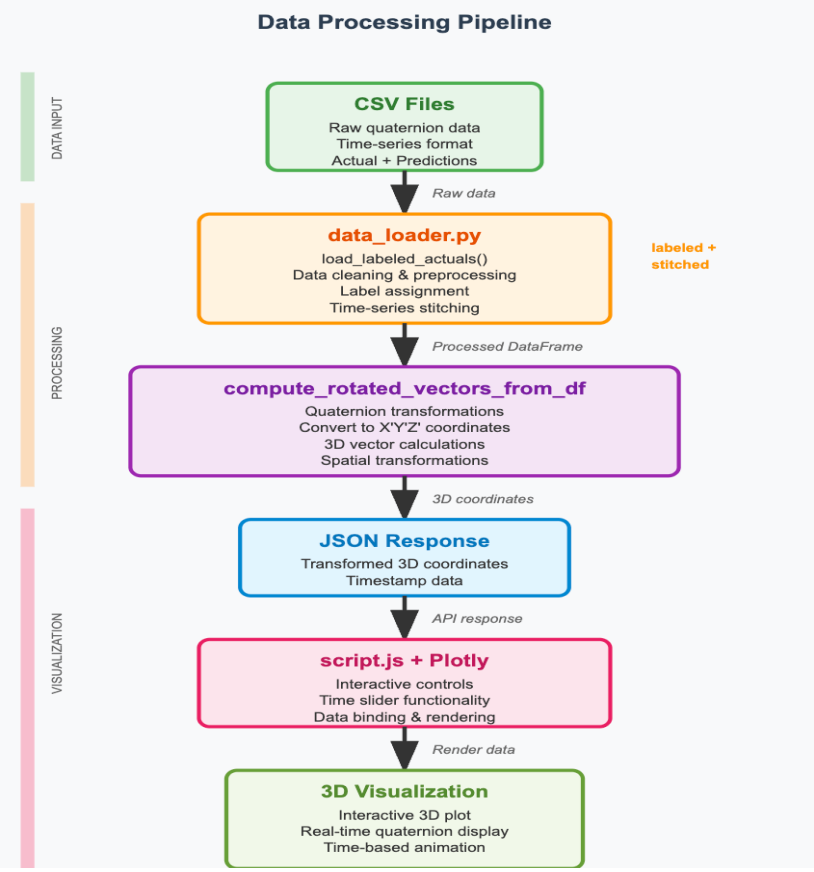
5.3 Quaternion Utilities (`quaternion_utils.py`)

- **Functions**:
  - `compute_rotated_vectors_from_df(df)`:
    - Applies each quaternion to unit basis vectors.
    - Rotates a cube defined in local body coordinates.
    - Returns a JSON-serializable result with X', Y', Z' and cube geometry.
- **Libraries Used**: `scipy.spatial.transform.Rotation`
- **Justification**:
  - Separates computational logic from data access or routing.
  - Testable, mathematically isolated, and scalable.

5.4 Frontend (`index.html` + `script.js`)

- **Features**:
  - 3D plot with Plotly.js.
  - Slider with real-time time updates.
  - Cube overlays for actual and predicted attitude.
  - Export buttons for CSV and PNG.
  - Timeline canvas showing source file color-coded by timestamp.
- **Design Decisions**:
  - Minimal dependencies (only Plotly via CDN).
  - Responsive design for browser compatibility.
  - JavaScript separation for logic modularity.

6. Data Flow Diagram



This diagram reflects a **pull-based data flow**, with the frontend requesting pre-processed JSON data from the server.

7. Export Features

- CSV Export Options:
  - Export all stitched data
  - Export actual-only rows
  - Export predicted-only rows
- PNG Export:
  - Captures current Plotly plot
  - Uses `Plotly.toImage()` for download without server dependency

8. UI Features

- Slider: Range input allows scrubbing through timestamps.
- Play/Pause Animation: Controlled via Spacebar.
- Speed Selector: User can control playback rate (1x, 2x, 5x).
- Prediction Toggle: Checkbox to hide/show predicted overlays.
- File Timeline Bar: Shows which file and type (actual/predicted) each point belongs to.

9. Design Decisions and Modularity

Code Structure

- **Backend and frontend separation** for clarity.
- **One function per file**, facilitating unit testing and easy replacement.
- **Stateless processing** — data is loaded fresh on each call, reducing caching complexity.

Function Signatures

- Each function takes only data and optional config as input. Example:
- `def load_labeled_actuals_and_predictions(data_folder="data", sample_limit=500):`

  Justification: Allows parameter tweaking during testing and production tuning.

- Reusable components (`compute_rotated_vectors_from_df(df)`) make no assumption about where the data is coming from.

General Flow

- Simplicity first: functional decomposition, no classes unless state retention is required.
- All visualization logic lives on the frontend — no rendering server-side.

10. Future Enhancements (Optional)

- Prediction error overlays with angle visualization.
- Support for real-time streaming data.
- UI drag-and-drop for CSV files.
- Toggleable cube scale.

11. Conclusion

This tool follows clean software architecture principles with clear interfaces, modular design, and end-user interactivity. It meets all the assignment requirements while leaving space for future enhancements. All design decisions were made to prioritize readability, flexibility, and testability.