VizLy: Self-Service Data Visualization Tool

Capstone Project Documentation

Author: Kurt Amodia

Date: October 2025

1. Problem Statement

In many data-driven organizations and research environments, the process of creating visual insights from raw datasets often requires technical expertise in data visualization tools or programming libraries such as Python's `Matplotlib`, `D3.js`, or R's `ggplot2`. This creates a dependency on data analysts or developers, limiting accessibility for non-technical users.

The lack of self-service visualization tools in small to medium organizations results in:

- Time-consuming data exploration processes.
- Bottlenecks in insight generation.
- Limited interactivity and flexibility in visual analytics.

Thus, there is a need for an easy-to-use platform that empowers users to create, manage, and visualize their datasets without requiring programming experience.

2. Proposed Solution

VizLy is a self-service web-based visualization tool that allows users to easily upload, process, and visualize their data. The platform provides a low-code environment where users can select datasets, define visualization parameters, and instantly generate charts and dashboards.

3. Objectives

The primary objectives of VizLy are as follows:

1. To design and implement an automated data processing pipeline capable of performing extraction, transformation, and loading (ETL) operations, ensuring data consistency, integrity, and readiness for visualization.

2. To develop an intuitive and modular user interface that enables non-technical users to perform data visualization tasks efficiently, while also supporting comparative performance analysis of multiple JavaScript visualization libraries—`Recharts`, `Chart.js`, and `Plotly.js`—in terms of usability, rendering performance, and bundle optimization.

3. To implement a real-time, interactive dashboard system that allows users to manage, monitor, and persist generated visualizations, enabling dynamic updates, data-driven insights, and seamless user interaction across multiple visualization modules.

4. System Architectural Design

4.1. Overview

VizLy follows a modular client-server architecture composed of distinct but interconnected layers designed for scalability and maintainability. The system components are defined as follows:

- **Frontend:** Developed using `NextJS` and `ShadcnUI`, providing an interactive, responsive, and component-based user interface that supports dynamic rendering of visualizations and dashboards.

- **Backend:** Implemented with `FastAPI`, serving as the core API for handling requests related to dataset management, chart operations, and dashboard updates. It enables efficient communication between the client and the database through RESTful endpoints.

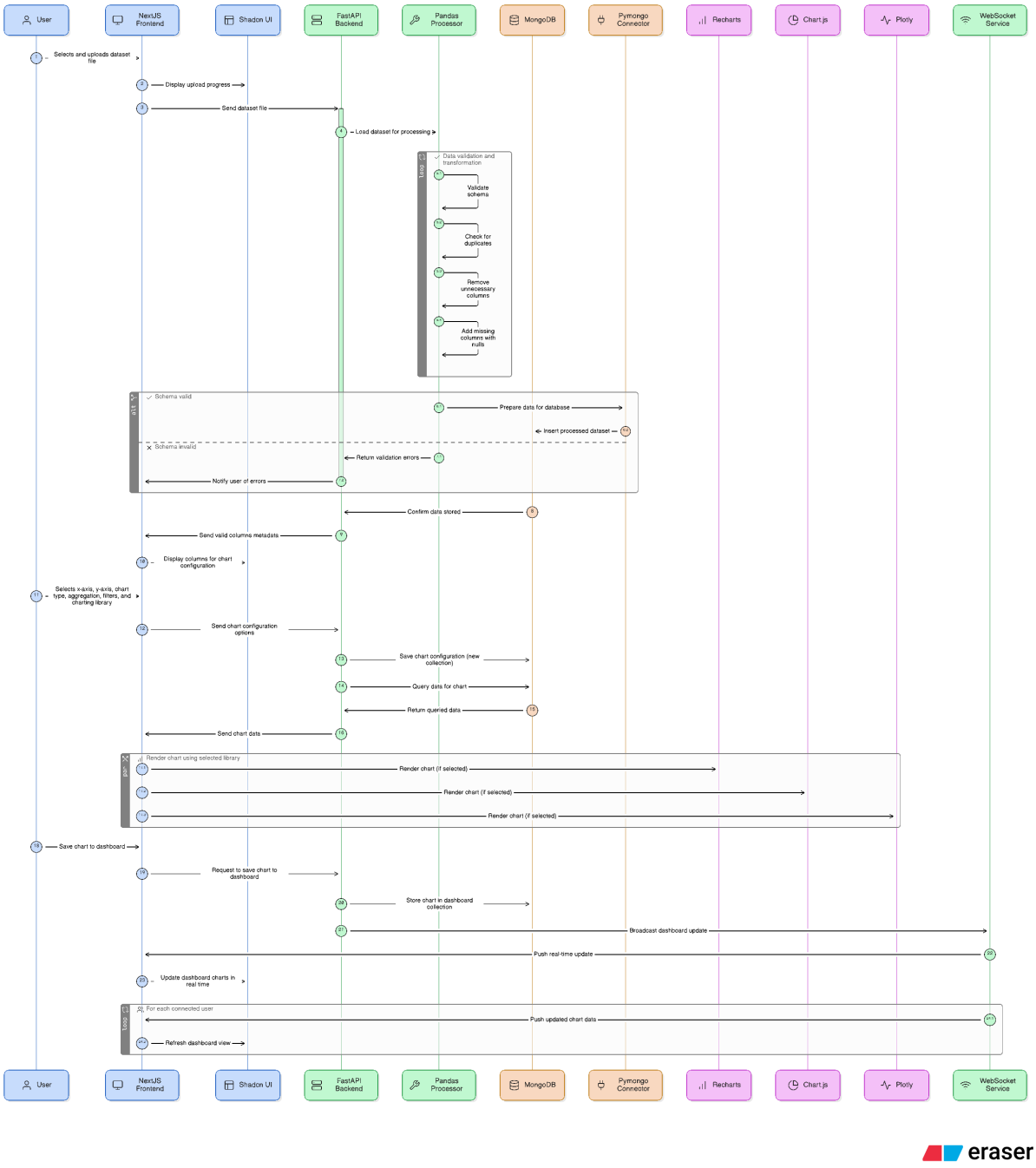
- **Data Processing:** Powered by the `Pandas` library, which performs data cleaning, transformation,validation tasks. It ensures accurate preprocessing of uploaded datasets before loading to the database.

- **Database:** Utilizes `MongoDB` to store user datasets, visualization configurations, and metadata related to saved charts. The database schema supports flexible document-based storage for both structured and semi-structured data.
- **Visualization Libraries:** Multiple JavaScript charting libraries are integrated to benchmark performance and visualization capabilities:
  - **Recharts:** A React-native library providing declarative and beginner-friendly chart creation using SVG rendering.
  - **Chart.js:** A lightweight library optimized for performance with Canvas-based rendering and minimal bundle size.
  - **Plotly.js:** A feature-rich library offering advanced analytics, 3D charts, and interactivity through WebGL rendering.

4.2. Data Flow

The data flow within VizLy is designed to ensure seamless processing and visualization generation. The workflow proceeds as follows:

1. **User uploads dataset:** The user imports a dataset (typically in CSV format) through the web interface.
2. **Data processing module:** The backend validates, parses, and preprocesses the dataset for compatibility with the visualization components.
3. **Visualization builder:** The system maps dataset columns to visualization parameters such as X and Y axes, chart type, and selected library.
4. **Chart rendering engine:** The visualization is generated using the selected JavaScript charting library.
5. **Dashboard display:** The generated charts are displayed in real-time and can be saved and modified by the user.



eraser

Figure 1: System Architecture Diagram of VizLy

5. Pydantic Models and Data Schemas

The backend system of VizLy utilizes the `Pydantic` library to define data models that ensure type safety, validation, and structured communication between the frontend and backend components. Each model represents the data schema for requests and responses handled by the `FastAPI` framework. These models serve as the foundation for data validation, serialization, and deserialization across API endpoints.

5.0.1 Dataset Model

The `Dataset` model defines the structure of records within uploaded datasets. It ensures schema consistency and proper data typing for statistical analysis and visualization.

- **upload_id:** Unique identifier for each uploaded dataset.
- **row_id:** Primary key reference for dataset rows.
- **model, year, region, color, transmission:** Descriptive fields used in analysis and grouping.
- **mileage_km, price_usd, sales_volume:** Quantitative attributes used in chart visualizations and aggregation computations.

5.0.2 Chart Model

The `Chart` model represents the configuration and metadata of a user-generated chart. It includes the chart type, axis configuration, aggregation function, and visualization library used. This model is primarily used in the chart creation, saving, and updating endpoints.

- **mode:** Determines the operational context (e.g., aggregated or dataset view).
- **upload_id:** Links the chart to a specific uploaded dataset.
- **chart_type:** Defines the visualization type (e.g., bar, line, pie).
- **x_axis, y_axis, agg_func:** Specify data relationships and aggregation behavior.
- **name:** Optional chart title for saved configurations.
- **chart_library:** Indicates the JavaScript visualization library used.
- **shareable:** Boolean flag to mark whether the chart can be publicly shared.

5.0.3 AggregateRequest Model

The `AggregateRequest` model defines the structure of requests sent to the aggregation endpoint, which performs mathematical computations such as sum, average, or count over a dataset column. It accepts optional filters such as `year_from` and `year_to` to limit the aggregation range.

- **upload_id:** Identifier of the uploaded dataset.
- **x_axis, y_axis:** Specify the columns to be used for aggregation.
- **agg_func:** Defines the aggregation function to be applied (default = `"sum"`).
- **year_from, year_to:** Optional range filters for year-based data aggregation.

5.0.4 Dashboard and DashboardUpdate Models

The `Dashboard` and `DashboardUpdate` models manage the configuration of visual dashboards within VizLy. Dashboards are composed of multiple charts and may be filtered by specific years or datasets.

- **mode:** Defines the dashboard context or view mode.
- **upload_id:** Associates the dashboard with a particular dataset.
- **chart_id:** References charts linked to the dashboard (optional for updates).
- **year_from, year_to:** Define the temporal range displayed on the dashboard.

The `DashboardUpdate` model is a lightweight version used when updating the date range only, ensuring minimal payload and faster API transactions.

5.0.5 Model Summary

These Pydantic models collectively ensure that all API requests and responses conform to a standardized format, reducing the risk of invalid data propagation throughout the system. By leveraging Pydantic's built-in validation mechanisms, VizLy maintains high reliability in data handling, allowing smooth integration between the FastAPI backend, the MongoDB database, and the Next.js frontend interface.

6. API Design and Endpoints

This section presents the design and implementation details of the API used in the system. All responses are returned in JSON format.

6.1. Dataset Endpoints

6.1.1 POST /api/dataset/upload

Description: Handles the upload of CSV datasets and stores associated column type metadata.

Request:

- **Method:** POST
- **Endpoint:** `/api/dataset/upload`
- **Content-Type:** `multipart/form-data`
- **Body Parameters:**
  - `file` (required) — CSV file to upload.

Responses:

- **200 OK:** Successful upload response in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.1.2 GET /api/dataset/all

Description: Retrieves all unique `upload_id` values from the database, representing previously uploaded datasets.

Responses:

- **200 OK:** Returns an array of `upload_id` values in JSON format.

6.1.3 GET /api/dataset/all/data

Description: Fetches all dataset records across all uploads.

Responses:

- **200 OK:** Returns all dataset records in JSON format.

6.1.4 GET /api/dataset/all/headers

Description: Retrieves all unique headers and the merged column types from all uploaded datasets.

Responses:

- **200 OK:** Returns headers and merged column types in JSON format.

6.1.5 GET /api/dataset/{upload_id}/data

Description: Fetches all dataset records for a specific upload identified by its `upload_id`.

Parameters:

- `upload_id` (path) — Unique identifier of the dataset.

Responses:

- **200 OK:** Returns dataset records in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.1.6 GET /api/dataset/{upload_id}/headers

Description: Retrieves the headers and column types for a specific dataset identified by `upload_id`.

Parameters:

- `upload_id` (path) — Unique identifier of the dataset.

Responses:

- **200 OK:** Returns headers and column types in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.2. Chart Endpoints

The Chart API handles operations related to chart generation, aggregation, storage, and retrieval. These endpoints allow users to create visual representations of dataset values and manage their saved charts within the system.

6.2.1 POST /api/chart/aggregate

Description: Performs data aggregation based on the parameters provided in the request body.

Request:

- **Method:** POST
- **Endpoint:** `/api/chart/aggregate`
- **Content-Type:** `application/json`
- **Body Parameters:**
  - `upload_id` — Dataset identifier.
  - `x_axis` — Column to be used for the X-axis.
  - `y_axis` — Column to be aggregated for the Y-axis.
  - `agg_func` — Aggregation function (e.g., `sum`, `mean`).
  - `year_from` — Starting year filter.
  - `year_to` — Ending year filter.

Responses:

- **200 OK:** Returns the aggregated data in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.2.2 POST /api/chart/save

Description: Saves a generated chart and its associated metadata to the database.

Request:

- **Method:** POST
- **Endpoint:** `/api/chart/save`
- **Content-Type:** `application/json`
- **Body Parameters:**
  - `mode`, `upload_id`, `chart_type`, `x_axis`, `y_axis`, `agg_func`, `year_from`, `year_to`, `name`, `chart_library`, `shareable`.

Responses:

- **200 OK:** Successful save response in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.2.3 PUT /api/chart/update/{chart_id}

Description: Updates an existing chart using its `chart_id`.

Parameters:

- `chart_id` (path) — Unique identifier of the chart.

Request Body:

- Same schema as `/api/chart/save`.

Responses:

- **200 OK:** Successful update response.
- **422 Unprocessable Entity:** Validation error.

6.2.4 GET /api/chart/saved/all

Description: Retrieves all saved charts available in the system.

Responses:

- **200 OK:** Returns all saved charts in JSON format.

6.2.5 GET /api/chart/shared/all

Description: Retrieves all chart IDs marked as shareable.

Responses:

- **200 OK:** Returns all shareable chart IDs in JSON format.

6.2.6 GET /api/chart/saved/{upload_id}

Description: Retrieves all saved charts associated with a specific `upload_id`.

Parameters:

- `upload_id` (path) — Unique dataset identifier.

Responses:

- **200 OK:** Returns all related saved charts.
- **422 Unprocessable Entity:** Validation error.

6.2.7 GET /api/chart/saved/chart/{chart_id}

Description: Retrieves a specific chart using its `chart_id`.

Parameters:

- `chart_id` (path) — Unique identifier of the chart.

Responses:

- **200 OK:** Returns chart details in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.2.8 DELETE /api/chart/delete/{chart_id}

Description: Deletes a chart record from the system based on its `chart_id`.

Parameters:

- `chart_id` (path) — Unique identifier of the chart.

Responses:

- **200 OK:** Successful deletion response.
- **422 Unprocessable Entity:** Validation error.

6.2.9 GET /api/chart/year-range

Description: Retrieves the minimum and maximum year values available within the uploaded dataset.

Parameters:

- `upload_id` (query, optional) — Dataset identifier to filter the range.

Responses:

- **200 OK:** Returns the year range in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.3. Dashboard Endpoints

The Dashboard API manages operations related to chart organization and visualization dashboards. It enables adding charts to dashboards, fetching existing dashboards, updating date ranges, and removing charts.

6.3.1 POST /api/dashboard/add

Description: Adds a chart to an existing dashboard based on `mode` and `upload_id`. If no dashboard exists, a new one is automatically created and associated with the chart.

Request:

- **Method:** POST
- **Endpoint:** `/api/dashboard/add`
- **Content-Type:** `application/json`
- **Body Parameters:**
  - `mode` — Operational mode of the dashboard.
  - `upload_id` — Dataset identifier.

- `chart_id` — Chart to be added.
- `year_from` — Starting year range.
- `year_to` — Ending year range.

Responses:

- **200 OK:** Successfully added chart to the dashboard.
- **422 Unprocessable Entity:** Validation error.

6.3.2 GET /api/dashboard/{mode}/{upload_id}

Description: Fetches a dashboard for a given `mode` and `upload_id`. If the dashboard exists, its chart details are automatically populated.

Parameters:

- `mode` (path) — Dashboard mode identifier.
- `upload_id` (path) — Dataset identifier (can be null).

Responses:

- **200 OK:** Returns the dashboard and chart details in JSON format.
- **422 Unprocessable Entity:** Validation error.

6.3.3 DELETE /api/dashboard/{dashboard_id}/{chart_id}

Description: Removes a specific chart from the dashboard based on its `dashboard_id` and `chart_id`.

Parameters:

- `dashboard_id` (path) — Unique identifier of the dashboard.
- `chart_id` (path) — Unique identifier of the chart to be removed.

Responses:

- **200 OK:** Chart successfully removed from the dashboard.
- **422 Unprocessable Entity:** Validation error.

6.3.4 PUT /api/dashboard/{dashboard_id}/date-range

Description: Updates the `year_from` and `year_to` fields of an existing dashboard to modify its displayed data range.

Parameters:

- `dashboard_id` (path) — Unique identifier of the dashboard.

Request Body:

- `year_from` — Starting year of the range.
- `year_to` — Ending year of the range.

Responses:

- **200 OK:** Dashboard date range successfully updated.
- **422 Unprocessable Entity:** Validation error.

7. Benchmark Results and Analysis

This section presents the benchmark results comparing three JavaScript charting libraries—**Recharts**, **Chart.js**, and **Plotly.js**—used in the development of the VizLy visualization system. The evaluation focuses on usability, extensiveness, rendering performance, and runtime efficiency under varying data loads.

7.1. Ease of Use

- **Recharts:** Highly beginner-friendly and React-native, making it well-suited for integration into modern React applications. It offers straightforward component-based chart creation with a strong and active developer community.
- **Chart.js:** Lightweight and easy to learn, Chart.js is ideal for simple to moderately complex charts. It has excellent documentation and community support, although it lacks built-in React components.
- **Plotly.js:** Provides feature-rich visualizations and powerful interactivity options but has a steeper learning curve and a heavier implementation footprint compared to the other two libraries.

7.2. Extensiveness

Each library offers varying degrees of extensibility and customization:

- **Recharts:** Provides a wide range of customizable chart types and supports data-driven rendering using React props and state.
- **Chart.js:** Offers an extensive plugin ecosystem, allowing users to extend chart functionalities through community or custom plugins.
- **Plotly.js:** The most extensive among the three, supporting advanced features such as 3D plots, interactive data selection, and real-time updates.

Documentation Sources:

- Recharts: https://recharts.github.io/en-US/api
- Chart.js: https://www.chartjs.org/docs/latest/
- Plotly.js: https://plotly.com/javascript/

7.3. Rendering and Bundle Size

- **Recharts:** Delivers moderate performance through efficient SVG rendering. Best suited for small to medium datasets.
- **Chart.js:** Uses an optimized Canvas rendering engine with minimal bundle size, providing fast rendering times even for larger datasets.
- **Plotly.js:** Incorporates WebGL for complex visualizations, enabling high interactivity but leading to slower initial load times and a larger bundle size.

7.4. Performance Metrics

Performance evaluation was conducted using multiple trials to ensure consistency:

- 20 iterations were averaged for datasets with fewer than 100 data points.
- 5 iterations were averaged for larger datasets.

Each library was benchmarked for its component initial mount render time on different sizes of data. The summarized results are presented in Table 1.

Table 1: Rendering Time on Initial Mount Comparison of Chart Libraries (in milliseconds)

Number of Data Points	**Recharts**	**Chart.js**	**Plotly.js**
<100	116.89	21.72	26.52
10,000	6278.96	124.78	329.20
50,000	30,791.38	550.44	1498.26

7.5. Conclusion

The benchmark comparison highlights trade-offs between simplicity, performance, and capability:

- For **simplicity** and seamless **React integration**: **Recharts** is recommended.
- For maximum **speed** and minimal **footprint**: **Chart.js** is the best choice.
- For advanced **analytics** and high **interactivity**: **Plotly.js** is most suitable.

Overall, these results guided the selection of the visualization library for VizLy, ensuring an optimal balance between system responsiveness, usability, and visualization capability.