PlotXpert

2023年10月5日

17:08

# Requirements

* Functional:
  + Support 2D XY Plotting
    - Able to plot a series of data points in the XY plane.
    - CRUD - Users can add/remove multiple data series to/from the plot.
    - Each data series can have its own visual properties (color, marker style, line style, etc.)
    - Support different types of plots, such as line, scatter.
    - The plot should auto-scale to fit the data.
    - Provide zooming and panning capabilities.
  + Support Curve operation and Computing
    - Basic: add, delete, multiply, division
    - CRUD.
    - Calculus: Used to solve derivatives and integrals of continuous functions, applied in scenarios such as object motion and mechanics analysis.
    - Interpolation and Fitting: Employed to fit curves or surfaces through a set of data points, facilitating the approximation of data.
    - Regression Analysis: Utilized to ascertain relationships between variables and identify optimal fitting curves.
    - Statistical Analysis: Encompassing descriptive statistics, inferential statistics, hypothesis testing, and more, used to extract insights from data.
    - Linear Algebra: Applied to solve systems of linear equations, perform matrix operations, and more.
    - Customized Extension Capabilities
  + Saving and exporting
    - Users can save their plot configurations for future use.
    - Export plots as common image files (e.g., .png, .jpeg) or vector graphics (e.g., .svg, .eps).
    - Export data in various formats (CSV, Excel, JSON, etc.)
  + Support extended application such as excel
    - Users can import data directly from Excel files.
    - Users can export data directly to Excel files.
    - Support for automation or integration with Excel via scripting or plugins.
  + Support Axis and Legend Configuration
    - Customizable axes labels.
    - Customizable axes scales (linear, logarithmic, etc.)
    - Customizable legend (position, style, visibility, etc.)
    - Gridlines can be toggled on/off.
  + Support layout state
    - Users can customize the layout of their plotting workspace.
    - Layout configurations can be saved and restored.
    - Users can have multiple layouts and switch between them as needed.

* Non-functional:
  + Performance
    - The web page must load in under 3 seconds under normal load conditions. The system must support 1000 transactions per second.
  + Maintainability
    - Cyclomatic complexity of any function must not exceed 10. The average time to fix a bug must be under 48 hours.
  + Scalability
    - The system must maintain a response time of under 5 seconds when the number of concurrent users increases by 50%. The system must be able to handle a 100% increase in data volume without degradation in performance.
  + Usability
    - A new user must be able to complete core tasks in under 10 minutes with minimal training. User error rates must not exceed 1% for key tasks.

# Solution(Architecture & Design & Technology Stack)

* Technology Stack
  + Language: JavaScript, Python, C++
  + Framework: Express.js
  + Environment: Node.js
  + Multithreaded programming to Highly concurrent server-side code
  + Deployment: Distributed web service applications - Azure
* Environment
  + IDE & Tool
    - VS, Visual Studio, Express.js, Node.js
    - Git & GitHub
  + Project
    - Initialize a new Node.js project using npm init.
    - Install Express.js (npm install express) and any other necessary packages for your backend.
    - Set up your C++ project structure (you might use CMake or Makefile based on your preferences).

* Architecture Style
  + Microservices Architecture
    - A single application is composed of multiple loosely coupled and independently deployable smaller services
  + Layered Architecture
    - Divides the application structure into distinct layers with specific responsibilities. Each layer interacts only with its neighboring layers, which ensures separation of concerns.
  + Event-Driven Architecture
  + Caching & Multithreaded Performance Optimization

* Serverless Architecture for scalability - Azure Function
* API Gateway
  + An API Gateway is a server that acts as an intermediary for requests from clients seeking resources from other servers or services. It handles API calls, routes requests to the appropriate services, and can perform various functions like request transformation, response transformation, authentication, and more.
* CI/CD Pipeline and DevOps - Azure DevOps
* Monitoring & Logging - Azure Monitor & Application Insights

* Overall Design
  + Logical
    - Layers
      * 1. Presentation
        + This layer handles the user interface and user experience.
        + It's responsible for presenting data visually to the end-user and collecting user inputs.
      * 2. AppServices
        + Connects the front-end (Presentation layer) with the back-end services (CoreEngine and Data).
        + Handles business logic and operations that aren't directly related to data manipulation or presentation.
      * 3. CoreEngine
        + This is where the primary computations, like curve fitting and mathematical transformations, happen.
        + Your C++ computational engine will reside here to ensure efficiency and performance.
      * 4. Data
        + Manages the data persistence, including saving, loading, and exporting functionalities.
        + Interfaces with databases or storage solutions.
    - Components
      * Client - Presentation
        + RecordDatas: These would be the datasets or collections of data points that users want to plot.
        + OutputType: Specifies the format for saving or exporting plots, e.g., PNG, SVG, or even data formats like CSV.
      * Services - AppServices & CoreEngine
        + Presentation: Handles how data is visually presented to users. It will use the CoreEngine to fetch processed data and display it in an easily digestible format.
        + Computation: Interfaces with the CoreEngine, especially the C++ computational module, to perform the necessary mathematical and plotting calculations.
        + FileManagement: Responsible for reading from and writing to files, especially for importing and exporting data.
        + LayoutState: Stores and retrieves user preferences about the plot's visual layout and appearance.
      * Plot - Presentation
        + DataSeries: Each unique collection of data points that a user wants to plot, with attributes like color, marker style, etc.
        + Canvas: The visual area where data points are plotted. It will handle the coordinate system, axes, gridlines, zooming, and panning functionalities.
    - Principles
      * Separation of Concerns (SoC): Each component or module should have a distinct responsibility. This principle is inherent in the layered architecture you're adopting.

* Single Responsibility Principle (SRP): Each class should have only one reason to change. This means that a class should only have one responsibility or job.

* Open/Closed Principle: Software entities should be open for extension but closed for modification. This allows you to add new functionality without changing existing code.

* Liskov Substitution Principle: Objects of a superclass shall be replaceable with objects of a subclass without affecting the correctness of the program.

* Interface Segregation Principle: A class should not be forced to implement interfaces it doesn't use.

* Dependency Inversion Principle: Depend on abstractions, not on concrete implementations.

* DRY (Don't Repeat Yourself): Avoid redundancy in your code.

* YAGNI (You Aren't Gonna Need It): Don't add functionality until it's necessary.

* KISS (Keep It Simple, Stupid): Keep your design and code as simple as possible.

* Patterns
  + Factory Pattern: Useful if you need to create objects based on certain criteria. For instance, you may have different methods of importing data (from CSV, Excel, etc.) and can use a factory to produce the right importer.

* Singleton Pattern: Ensures a class has only one instance and provides a global point to access it. This might be useful for certain services that you only want instantiated once, like a configuration manager.

* Strategy Pattern: Useful when you have multiple algorithms to perform a specific action. This can be beneficial for different plotting strategies or data processing methods.

* Observer Pattern: Especially useful in GUI applications, where changes in one component (e.g., data) need to be reflected in another (e.g., a plot).

* Decorator Pattern: Provides a way to add responsibilities to objects dynamically. Can be useful for adding features to your plots or data series without modifying their structure.

* Command Pattern: Encapsulate a request as an object, allowing parameterization with different requests. This can be handy if you implement features like "Undo" or "Redo".

* Adapter Pattern: Convert the interface of a class into another interface clients expect. Useful for integrating with third-party libraries or when evolving old components.

* State Pattern: Allows an object to change its behavior when its internal state changes. Useful for things like managing the different layout states.

* Composite Pattern: Compose objects into tree structures to represent part-whole hierarchies. Useful if you have nested plots or hierarchical data structures.

* Memento Pattern: Useful for implementing features like "Undo" by capturing and restoring an object's internal state.

* Process
* Data
* UML (PlantUML)

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam class {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

skinparam interface {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

package "Layer Hierarchy"{

package Presentation

package AppServices

package CoreEngine

package Data

}

Presentation ..> AppServices

AppServices ..> CoreEngine

CoreEngine ..> Data

@enduml

* CoreEngine(C++, Strategy pattern)
  + UML (PlantUML)

@startuml

' Enum

enum OperationType {

Unary,

Binary,

CustomExtend

}

' CoreEngine and its related components

class CoreEngine {

- static instance: CoreEngine

- operationRegistrant: OperationRegistrant

- operationCache: Cache<OperationType, IOperation>

- operandCache: Cache<std::string, IPointSeries>

- resultCache: Cache<string, OperationResult>

+ static getInstance(): CoreEngine

+ execute(context: IOperationContext): OperationResult

+ getContext(type: OperationType): IOperationContext

+ releaseContext(context: IOperationContext): void

}

CoreEngine \*-down- OperationRegistrant

CoreEngine \*-down- Cache

' Context interfaces and classes

interface IOperationContext <<interface>> {

+ getOperationType(): OperationType

+ getData(): vector<IPointSeries>

}

class UnaryOperationContext {

-operationType: OperationType

-data: IPointSeries

+getOperationType(): OperationType

+getData(): IPointSeries

}

class BinaryOperationContext {

-operationType: OperationType

-data1: IPointSeries

-data2: IPointSeries

+getOperationType(): OperationType

+getData1(): IPointSeries

+getData2(): IPointSeries

}

class CustomExtendOperationContext<T...> {

-operationType: OperationType

-data: vector<CustomPointSeries<T...>>

+getOperationType(): OperationType

+getData(): vector<CustomPointSeries<T...>>

}

IOperationContext <|.. UnaryOperationContext

IOperationContext <|.. BinaryOperationContext

IOperationContext <|.. CustomExtendOperationContext

' Data interfaces and classes

interface IPointSeries <<interface>> {

+ getPoints(): vector<Point>

}

class 1DPointSeries {

- x: double

+getPoints(): vector<Point>

}

class 2DPointSeries {

- x: double

- y: double

+getPoints(): vector<Point>

}

class 3DPointSeries {

- x: double

- y: double

- z: double

+getPoints(): vector<Point>

}

class CustomPointSeries<T...> {

- points: vector<CustomPoint<T...>>

+getPoints(): vector<Point>

}

IPointSeries <|.. 1DPointSeries

IPointSeries <|.. 2DPointSeries

IPointSeries <|.. 3DPointSeries

IPointSeries <|.. CustomPointSeries

class OperationResult{

-resultPoints: IPointSeries\*

-isSuccess: bool

-message: std::string

}

' Operation interfaces and classes

interface IOperation <<interface>> {

+ execute(data: IPointSeries): OperationResult

}

interface IUnaryOperation <<interface>> extends IOperation {

+execute(data: IPointSeries): OperationResult

}

interface IBinaryOperation <<interface>> extends IOperation {

+execute(data1: IPointSeries, data2: IPointSeries): OperationResult

}

interface ICustomExtendOperation<T...> <<interface>> extends IOperation {

+execute(data: vector<CustomPointSeries<T...>>): OperationResult

-m\_functions: vector<Function>

}

class OperationRegistrant {

- operations: map<OperationType, IOperation\*>

- static instance: OperationRegistrant

+ registerOperation(type: OperationType, operation: IOperation): void

+ getOperation(type: OperationType): IOperation

}

' Utility classes for caching

class Cache {

- items: map<K, V>

- mutex: Mutex

+ getItem(key: K): V

+ putItem(key: K, value: V): void

}

' Relationships

CoreEngine --> IOperationContext : uses > execute

OperationRegistrant --> IOperation : provides

@enduml

* Functionalities:
  + Calculus: Used to solve derivatives and integrals of continuous functions, applied in scenarios such as object motion and mechanics analysis.
  + Approximation: Interpolation and Fitting, employed to fit curves or surfaces through a set of data points, facilitating the approximation of data.
  + Regression Analysis: Utilized to ascertain relationships between variables and identify optimal fitting curves.
  + Statistical Analysis: Encompassing descriptive statistics, inferential statistics, hypothesis testing, and more, used to extract insights from data.
  + Linear Algebra: Applied to solve systems of linear equations, perform matrix operations, and more.
  + Customized Extension Capabilities
* Well defined Interfaces
  + How to transfer data? JSON or XML?
  + Foreign Function Interface (FFI) with Node.js

* AppServices(JS, MVC and Microservice(SOA) architecture)
  + UML (PlantUML)

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam class {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

skinparam interface {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

class DisplayController {

+ renderPlot

+ handleInteraction

+ updateSetting

+ notifyMessage

}

interface IDisplayService <<interface>>{

}

class DataProcessService{

}

class VisualizationSettingService{

}

class InteractionHandleService{

}

class NotificationService{

}

DisplayController -down..> IDisplayService : uses

DataProcessService -up..|> IDisplayService

VisualizationSettingService -up..|> IDisplayService

InteractionHandleService -up..|> IDisplayService

NotificationService -up..|> IDisplayService

class ComputaionController{

+ executeComuptation

+ getResults

+ manageError

+ manageComputationState

+ handleTasks

}

interface IComputationService <<interface>>{

}

class ComputationExecutionService{

}

class ResultsRetrievalService{

}

class ErrorHandlingService{

}

class StateManagementService{

}

class TaskQueuingService{

}

ComputaionController -down..> IComputationService : uses

ComputationExecutionService -up..|> IComputationService

ResultsRetrievalService-up..|> IComputationService

ErrorHandlingService -up..|> IComputationService

StateManagementService -up..|> IComputationService

TaskQueuingService -up..|> IComputationService

@enduml

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam class {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

skinparam interface {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

class FileManagementController{

+ loadFile

+ saveFile

+ manageStorage

+ manageFileVersion

}

interface IFileManagementService <<interface>>{

}

class FileReadWriteService{

}

class FileFormatConversionService{

}

class StorageManagementService{

}

class FileMetadataService{

}

class FileErrorHandlingService{

}

FileManagementController -down..> IFileManagementService : uses

FileReadWriteService -up..|> IFileManagementService

FileFormatConversionService -up..|> IFileManagementService

StorageManagementService -up..|> IFileManagementService

FileMetadataService -up..|> IFileManagementService

FileErrorHandlingService -up..|> IFileManagementService

class SnapshotController{

+ createSnapshot

+ applySnapshot

+ manageSnapshot

}

interface ISnapshotService <<interface>>{

}

class SnapshotCreationService{

}

class SnapshotRetrievalService{

}

class SnapshotManagementService{

}

class SnapshotMetadataService{

}

SnapshotController -down..> ISnapshotService : uses

SnapshotCreationService -up..|> ISnapshotService

SnapshotRetrievalService -up..|> ISnapshotService

SnapshotManagementService -up..|> ISnapshotService

SnapshotMetadataService -up..|> ISnapshotService

@enduml

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam class {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

skinparam interface {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

class CustomizationController{

+ addCustomFunction

+ removeCustomFunction

+ triggerEvent

+ managePluginCofig

}

interface ICustomPlugin <<interface>>{

+ initialize

+ enable

+ disable

+ destory

}

class EventBus{

+ emit

+ on

+ off

}

class PluginRegistry{

+ registerPlugin

+ removePlugin

+ getPlugin

}

class PluginManager{

+ saveConfig

+ getConfig

+ updateConfig

}

CustomizationController -down..> ICustomPlugin : uses

CustomizationController -down..> EventBus: uses

CustomizationController -down..> PluginRegistry : uses

CustomizationController -down..> PluginManager : uses

EventBus -left..> ICustomPlugin : uses

@enduml

* Functionalities
  + Presentation Functionalities
    - Data Formatting and Structuring:
      * Converts raw computation results into structured data suitable for rendering. This might involve turning raw coordinates into formatted plot points or data series.
      * Ensures data is packaged with relevant metadata like axes labels, data series names, and more.

* Interaction Management:
  + Processes user interactions like zooming, panning, clicking on data points, etc.
  + Determines and sends back relevant information to the UI when users interact with plots. For instance, when a user hovers over a data point, the service can provide detailed info about that point.

* Collaborates with other services (like FileManagementService for importing/exporting visual settings or LayoutStateService for user preferences).
* Ensures a seamless flow of data and operations between different application components.

* Visualization Settings:
  + Applies user-selected or default visual themes to the data plots.
  + Manages look and feel attributes like colors, line types, grid visibility, etc.
  + Enables dynamic theme adjustments as per user preferences.

* Remembers and applies user-defined visualization settings. This could be in terms of default plot types, preferred color schemes, or even saved layouts.
* Offers facilities to update these settings dynamically.

* Notification Message:
  + Provides feedback messages to the UI. This can include success messages, error alerts, or informative tooltips.
  + Guides users with suggestions or helps messages, especially when dealing with complex plots or data sets.

* Computation Functionalities
  + Computation Execution:
    - Act as an intermediary between the presentation layer and the CoreEngine.
    - Initiate and manage various computations based on user actions or automatic triggers.

* Computation Results Retrieval:
  + Retrieve and interpret results from the CoreEngine after computations.
  + Convert computation results into a suitable format that can be further processed by other services or directly sent to the presentation layer.

* Computation Error Handling:
  + Manage errors arising from computational tasks, including exceptions, data-related issues, or errors from the CoreEngine.
  + Provide meaningful error messages or feedback to the presentation layer.

* State Management:
  + Manage and store interim computational states to enable functionalities such as pause/resume computations or to quickly retrieve results for recurrent computations.

* Task Queuing & Management:
  + Queue computational tasks if multiple tasks are requested in rapid succession.
  + Manage priorities and execution order of computational tasks.
* FileManagement Functionalities
  + File Reading and Writing:
    - Read files from a specified location.
    - Write data to files in a specified location.
    - Support for various file formats (e.g., CSV, XLSX, JSON).

* File Format Conversion:
  + Convert between supported file formats.
  + Provide error handling for unsupported formats.

* Storage Management:
  + Manage file locations (local, cloud, network).
  + Organize files in folders or directories.
  + Handle file backups.

* File Metadata Handling:
  + Retrieve and store file metadata (e.g., creation date, last modified, file size).
  + Provide methods to search files based on metadata.

* File Versioning:
  + Keep track of different versions of a file.
  + Provide methods to retrieve previous file versions.

* Error Handling:
  + Handle file read/write errors.
  + Handle unsupported file format errors.
  + Manage issues related to storage access or permissions.
* LayoutState Functionalities
  + Snapshot Creation:
    - Capture the current state of the layout.
    - Allow for naming or tagging the snapshot for easier identification.

* Snapshot Retrieval:
  + List all available snapshots.
  + Load a specific snapshot by name or ID.

* Snapshot Management:
  + Delete specific snapshots.
  + Rename or re-tag snapshots.
  + Compare two snapshots to highlight differences.

* Metadata Management:
  + Store metadata with each snapshot (e.g., timestamp of creation, user notes).
  + Retrieve snapshot metadata.

* Customization Functionalities
  + Plugin Interface:
    - Define a clear interface for custom plugins. Every custom function or module should adhere to this interface.

* Plugin Registry:
  + Create a registry that keeps track of available plugins and allows easy addition or removal of plugins.

* Plugin Lifecycle Management:
  + Provide a mechanism to initialize, enable, disable, or destroy a plugin.

* Events and Hooks:
  + Allow plugins to listen to certain events and react to them. This can be achieved through an event-driven architecture where plugins can register their own event listeners.

* Configuration and Metadata:
  + Each plugin might have its own configuration. Provide a mechanism to store and manage these configurations.

* API Exposure:
  + If plugins need to interact with other parts of the application, expose necessary APIs to them.

* Well defined Interfaces
* Implementation

* Presentation
  + UML (PlantUML)

@startuml

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam class {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

skinparam interface {

BackgroundColor White

BorderColor Black

FontName Helvetica

FontSize 16

AttributeFontSize 14

}

class Client{

- inputData: Data

- plotPreferences: PlotPreferences

+ uploadData(data: Data)

+ setPlotPreferences(preferences: PlotPreferences)

+ interactWithPlot(interaction: ClientInteraction)

+ saveExportShare()

}

class PlotManager{

+ process(data: Data, preferences: PlotPreferences): RenderedPlot

+ computePlotData(data: Data): ComputedData

+ renderPlot(computedData: ComputedData, preferences: PlotPreferences): RenderedPlot

+ adjustPlot(interaction: ClientInteraction)

}

class ClientInteraction{

- type: InteractionType

- value: any

}

class Data{

- rawData: array double

}

class Plot{

- visualization: Image

}

class PlotPreference{

- xAxisLabel: string

- yAxisLabel: string

- plotType: PlotType

+ validate(): bool

}

Client -down..> PlotManager : Requests Plot

Client -down..> Data : Uploads

Client -down..> PlotPreference : uses

Client -down..> ClientInteraction : Performs

PlotManager -down..> Data : uses

PlotManager -down..> Plot : uses

PlotManager -down..> PlotPreference : uses

PlotManager -down..> ClientInteraction : uses

@enduml

@enduml

@startuml

' Hide the Attributeicons for classes and interfaces

skinparam classAttributeIconSize 0

skinparam classStereotypeIconSize 0

' Hide the icons for classes and interfaces

hide circle

' Customize the style

skinparam enum {

FontSize 20

}

enum InteractionType{

Z O O M

P A N

F I T

}

enum PlotType{

L I N E

B A R

S C A T T E R

P I E

}

@enduml

* 1. Client Input:
  + Data Input Module:
    - A simple form or file upload option where users can submit their data.
    - Data validation: Check if the uploaded data is in the right format and meets certain criteria.
    - Help or tooltip: Provide a sample format or guidelines for how the data should be structured.
  + Preferences Input:
    - Type of Plot: Dropdown to select between different plots like line, bar, scatter, etc.
    - Color Theme: Color picker or predefined color palette selections.
    - Axis Labels: Text fields to input x-axis and y-axis labels.
    - Other Preferences: Gridlines on/off, legends on/off, etc.
* 2. Processing Request:
  + Data Parsing:
    - Convert uploaded data into a standardized format.
    - Error handling: If data can't be parsed, notify the user.
  + Computation:
    - The request is sent to the backend or computation module to process the data based on the plot type.
    - Transformation and aggregation might be needed, depending on the plot type.
* 3. Rendering Plot:
  + Visualization Module:
    - Use a plotting library (like Chart.js, D3.js) to render the plot based on the provided data and user preferences.
    - Adjust visualization settings as per user preferences (color theme, axis labels, etc.).
  + Dynamic Interaction:
    - User can hover over data points to see details.
    - Panning and zooming functionalities are enabled.
* 4. Feedback to Client:
  + Success:
    - Once the plot is rendered, a success message or sound notification is given to the user.
  + Errors:
    - If there's an issue at any step, an error message is displayed, explaining the problem and suggesting potential solutions.
* 5. User Adjustments:
  + Interactive Controls:
    - Sliders or dropdowns to dynamically adjust the plot, like changing its type, adjusting axes limits, etc.
    - Buttons to save the plot, export as an image or PDF, or share with others.
* Flow:
  + User accesses the PlotXpert web interface.
  + User uploads data and sets plot preferences.
  + PlotXpert processes and validates the data.
  + PlotXpert computes necessary plot data (interactions, aggregations).
  + Plot is rendered based on user's preferences.
  + User interacts with the rendered plot, possibly making further adjustments.
  + User can then save, export, or share the plot.

# Testing

# Deployment & DevOps

# Project Management