

IMWEBs Viewer Technical Report

Desktop Application Development Using Vue, Python, and Tauri



January 27, 2025

ECCC

[Company address]

Table of Contents

[Abstract 3](#_Toc188041406)

[Introduction 4](#_Toc188041407)

[Background 4](#_Toc188041408)

[Objective 4](#_Toc188041409)

[Scope 4](#_Toc188041410)

[Methodology 4](#_Toc188041411)

[Development Tools 4](#_Toc188041412)

[Application Architecture 5](#_Toc188041413)

[Results and Implementation 5](#_Toc188041414)

[Backend 5](#_Toc188041415)

[Frontend 6](#_Toc188041416)

[App: 6](#_Toc188041417)

[Project & Table: 7](#_Toc188041418)

[Graph: 9](#_Toc188041419)

[Column Dropdown Component: 10](#_Toc188041420)

[Database Dropdown Component: 11](#_Toc188041421)

[Export Configuration Component: 12](#_Toc188041422)

[Export Table and Stats Component: 13](#_Toc188041423)

[Folder Tree Component: 14](#_Toc188041424)

[Interval Dropdown Component: 16](#_Toc188041425)

[Message Box Component: 17](#_Toc188041426)

[Selection Component: 18](#_Toc188041427)

[Statistics Dropdown Component: 19](#_Toc188041428)

[Security 20](#_Toc188041429)

[Cross-Site Scripting (XSS) and the Importance of Input Sanitization in State Management 20](#_Toc188041430)

[Migration from Electron to Tauri 21](#_Toc188041431)

[Initial Setup 21](#_Toc188041432)

[Challenges & Solutions 21](#_Toc188041433)

[Performance Gains 21](#_Toc188041434)

[Disadvantages of Tauri 22](#_Toc188041435)

[Conclusion 22](#_Toc188041436)

[GitHub Actions for Release Bundles 22](#_Toc188041437)

[Overview 22](#_Toc188041438)

[Workflow File Setup 22](#_Toc188041439)

[Challenges 23](#_Toc188041440)

[Workflow Highlights 23](#_Toc188041441)

[Current Status 24](#_Toc188041442)

[Conclusion 24](#_Toc188041443)

[Discussion 24](#_Toc188041444)

[Challenges Encountered & Solutions 24](#_Toc188041445)

[Recommendations for Future Work 25](#_Toc188041446)

[References 25](#_Toc188041447)

# Abstract

This technical report presents the design, development, and implementation of a desktop application for visualizing watershed management data using Vue for the frontend and Python for the backend. The application was initially developed using Electron but later migrated to Tauri for improved performance and native system integration. The report details the project's objectives, the development environment, the technologies used, the application structure, security and challenges encountered during the development process. Finally, recommendations for future improvements and enhancements are provided.

# 

# Introduction

## Background

The IMWEBs Viewer project is part of an effort to develop a comprehensive tool for visualizing and analyzing watershed data. The application needs to be accessible as a desktop app with a user-friendly interface for loading, displaying, and interacting with data. The backend must support data processing and analysis, while the frontend should provide visualization and user controls.

## Objective

To develop a cross-platform desktop application using Vue for the frontend and Python for backend services. Initially, the app was developed using Electron, but due to performance considerations and the need for lightweight deployment, Tauri was chosen as the framework for final implementation.

## Scope

The report covers:

The development tools and frameworks used.

The software architecture of the desktop application.

A step-by-step guide to implementing key features.

Challenges encountered during migration from Electron to Tauri.

Discussion regarding general challenges.

# Methodology

## Development Tools

Frontend: Vue.js (JavaScript framework)

Backend: Python with Flask (API server)

Packaging Framework: Electron (initial) and Tauri (final)

State Management: Vuex for reactive state management

Database: SQLite for local data storage

Version Control: Git and GitHub for source control

Build Tools: Webpack for Vue, Tauri build scripts

## Application Architecture

The desktop application follows a client-server architecture:

Frontend: Vue.js handles the UI, folder tree structure, and user interactions.

Backend: Python (Flask) provides REST APIs to process data, manage database queries, and serve computational functions such as statistical analysis.

Bridge: Electron (initially) and Tauri (for final version) were used to bridge communication between the frontend and backend while packaging the app into a native desktop executable.

# Results and Implementation

## Backend

The Backend Python Flask API is designed to facilitate data retrieval, exportation, and database interaction in a web-based tool. Key features include:

* User Authentication:
  + The /api/login endpoint authenticates users using a username and hashed password. If the provided credentials match the stored admin credentials, a JWT access token is generated and returned. This token is used to authenticate further requests to protected endpoints. The system ensures security by storing passwords in a hashed format and verifying them securely.
* User Logout and Token Revocation:
  + The /api/logout endpoint revokes the JWT token used in the request, effectively logging the user out. Upon logout, the token’s unique identifier (JTI) is added to a server-side revocation list, ensuring that the token cannot be reused. This mechanism ensures secure and proper session termination.
* Token Verification:
  + The /api/verify-token endpoint checks the validity of the JWT token included in the request headers. If the token is present and valid, the API returns a success response indicating that the token is authorized. Otherwise, it returns an unauthorized status, providing a simple mechanism for frontend applications to check login state.
* Folder Upload:
  + The /api/upload\_folder endpoint allows authenticated users to upload folders containing multiple files to the server. The files are received via a multipart/form-data request and saved to the predefined server directory structure. Each uploaded file is placed into its corresponding subdirectory, and any missing folder paths are automatically created.
* Data Retrieval:
  + The /api/get\_data endpoint fetches data based on user-provided parameters, including database path, table name, selected columns, IDs, and date ranges. The service also supports time-based aggregation and statistical analysis for time-series data.
* Data Export:
  + The /api/export\_data endpoint allows users to export data and statistics in CSV or text format, with options to select the specific data and statistical results to include in the file. Additionally, users can export an Excel file with an embedded graph, create an image file (png, jpg, jpeg, svg, or pdf) containing the graph or shapefiles of the selected data and geospatial data.
* Table and File Listing:
  + The /api/get\_tables endpoint retrieves table names from a given database path, and the /api/list\_files endpoint lists files and folders in a specified directory.
* Table Details:
  + The /api/get\_table\_details endpoint provides metadata for multiple tables from different or same database, including column names, time ranges, and available IDs. It also determines the type of time interval (daily, monthly, yearly) for time-series data.
* Caching and Performance:
  + Caching is implemented for several endpoints using Flask’s caching mechanism, storing query results for 5 minutes to optimize performance.
* Server and Cache Management:
  + The API includes administrative routes for shutting down the server only if app is running as a standalone executable (/api/shutdown), checking server status (/api/health) and clearing the cache (/api/clear\_cache).
* Bidirectional Mapping:
  + The mapping between real names and aliases is stored in the lookup.db3 database, where each table and column has a corresponding alias. The API loads this mapping on demand and uses it in both directions:
  + Real to Alias: When sending data back to the frontend.
  + Alias to Real: When processing user requests and interacting with the database.
* Dynamic Query Translation:
  + When querying the database, the API translates aliases in the user request into the real table and column names. After fetching the results, the API converts the real names in the response back to aliases before returning the data to the user. This ensures users can interact with the API using simple, human-readable names while the underlying database uses the correct schema names.
* Request Validation and Security Checks:
  + All incoming requests are validated using the validate\_request\_args function, which utilizes the Cerberus library to enforce schemas for required fields, types, and constraints.
  + The validation process includes additional security checks for detecting potential path traversal and SQL injection attempts, ensuring robust protection against malicious input.
  + Each API endpoint has a specific validation schema tailored to its functionality, making the system flexible and secure.
  + By combining validation with detailed error responses, the API guides users in providing correct parameters, improving user experience and system reliability.
* Geospatial Data Retrieval:
  + The /api/geospatial endpoint returns GeoJSON data, TIFF image URLs, bounds, and center coordinates based on user-provided parameters.
  + The processed geospatial data is retrieved and returned in JSON format.
* GeoTIFF Image Serving:
  + The /api/geotiff/<path:filename> endpoint allows serving TIFF/PNG files dynamically.
  + The TIFF file is then returned with a PNG MIME type for visualization compatibility.
* GeoJSON Color Retrieval:
  + /api/get\_geojson\_colors: Retrieves color configurations for GeoJSON data based on user-specified parameters. This process involves fetching raw data, applying statistical calculations, binning values into quantile-based categories, and assigning colors to individual IDs.
* Map Exportation:
  + The /api/export\_map endpoint allows users to export map images.
  + The endpoint receives an uploaded image and associated form data with image, export filename, format and path.
  + The validated image is processed and saved, with the generated file path returned as a JSON response.

API Routes: <https://github.com/shahviransh/ECCC-IMWEBs-Viewer/blob/main/backend/routes.py>

API Services: <https://github.com/shahviransh/ECCC-IMWEBs-Viewer/blob/main/backend/services.py>

These services interact with an SQLite database and support data transformation, making the API suitable for managing time-series data and statistical analysis.

## Frontend

### App:

The main page of the IMWEBs Viewer application serves as the central hub for user interaction and navigation. It provides a structured interface for managing data and accessing the core features of the application. The page is divided into three key sections: the Top Bar, Taskbar, and Main Content.

Key Features:

**1. Top Bar**

* Displays the application title, "IMWEBs Viewer", for clear identification.
* Includes a theme toggle button to switch between light and dark themes.
* The button visually updates to display a sun (🌞) for light mode or a moon (🌜) for dark mode, reflecting the active theme.

**2. Taskbar**

* Left Section:
  + Provides a button for folder selection (📁 Select Folder) to specify the working directory, allowing the user to select the model folder location.
  + Displays the selected folder path for user reference.
  + Includes zoom control buttons for Map & Graph:
  + Zoom In: Increases the zoom level within defined limits.
  + Zoom Out: Decreases the zoom level within defined limits.
  + Reset Zoom: Resets the zoom to its default state.
* Right Section:
  + Contains navigation buttons for accessing different pages of the application: Project, Table, Graph, Map, Calibration, BMP, Tools, and Help.
  + Highlights the currently active page for user clarity and updates the application state accordingly.

**3. Main Content**

* Displays the core content of the application based on the selected page.
* Utilizes Vue Router’s <router-view> to dynamically render content as users navigate between pages.

**4. MessageBox Component**

* A placeholder MessageBox component is included for potential notifications or user messages.

### Project & Table:

The Project and Table pages of the IMWEBs Viewer application provide users with tools to interact with datasets through navigation, filtering, visualization, and export functionality. These pages are central to managing and analyzing project data and statistics.

Key Features:

**1. Folder and Column Navigation**

* Database Dropdown: Allows users to select a database for data operations.
* Column Dropdown: Enables users to choose specific columns from the selected table for visualization and export.

**2. Settings Panel**

This panel organizes tools for configuring data operations:

* Selection Component: For filtering data by specific IDs or criteria.
* Interval Dropdown: Sets the aggregation level, such as daily, monthly, or yearly intervals.
* Statistics Dropdown: Provides options to select statistical metrics like mean, median, or sum.
* Export Configuration: Configures export options, including file format, path, and columns.
* Fetch Data: Retrieves filtered data from the backend.
* Export Data: Exports the data and statistics to files based on user-defined configurations.

**3. Main View**

Displays data and statistics in a structured layout:

* Table Container: Renders the dataset as a scrollable table. Implements a "Load More" button to handle large datasets incrementally, improving performance.
* Statistics Container: Displays statistical results in a similar scrollable table for easy comparison.
* Dynamic Data Loading: The methods ensure efficient data rendering by initially displaying a limited number of rows and incrementally loading more on user request. The properties determine whether additional rows can be loaded, dynamically showing or hiding the "Load More" button.
* Data Fetching: The fetchData method sends a GET request to the backend API with parameters such as selected database, table, columns, IDs, and date range. Displays success or error messages using a message queue.
* Data Export: The exportData method sends an export request to the backend, allowing users to save filtered data and statistics in formats such as CSV or Excel. Includes user-defined configurations for file paths, names, and additional options.

**4. Backend Integration**

* Fetch Data and Export Data requests use axios to communicate with a backend API, passing necessary parameters for processing.
* Responses include data for tables and statistics, which are rendered dynamically on the frontend.

### Graph:

The Graph page in the IMWEBs Viewer application enables users to visualize data dynamically in the form of graphs. It integrates advanced charting capabilities using ECharts and Vue-ECharts, making it a core component for data analysis and presentation.

Key Features:

**1. Folder and Column Navigation**

* Database Dropdown: Lets users select the database containing the data to be visualized.
* Column Dropdown: Allows selection of specific columns from the table for visualization.

**2. Settings Panel**

* Provides configuration options for:
* Selection: Enables filtering of data based on specific IDs or criteria.
* Interval Dropdown: Sets data aggregation intervals (e.g., daily, monthly, yearly).
* Statistics Dropdown: Configures statistical metrics for the data.
* Export Configuration: Sets parameters for exporting graphs and associated data.
* Fetch Data: Loads the data to be visualized in the graph.
* Export Data: Exports graph data in user-specified formats.

**3. Graph Display**

* Dynamic Charting: A graph is rendered using Vue-ECharts (<v-chart> component) with customizable options.
* Graph Types: Supports multiple types of charts (line, scatter, bar) with dual y-axis support for varied data ranges. Also, supports selection of parameters as combinations of graph types, such as Line & Scatter, Scatter & Bar, Bar & Line, and Bar, Line & Scatter.
* Tooltips: Display detailed information about data points.
* Legend: Scrollable and adjustable to handle multiple datasets.
* Data Zoom: Allows users to zoom in and out of specific sections of the graph.

**4. Functional Highlights**

* Legend Customization: Legends dynamically adjust to show column names and associated IDs.
* Multi-Axis Support: Columns requiring separate scaling are plotted on a secondary y-axis.
* Data Lookup: A preprocessing step maps each ID and date to its corresponding row for efficient chart rendering.
* X-axis: Displays dates or months, dynamically generated from the data.
* Y-axes: Include primary and secondary axes for varied data ranges.
* The graph data is processed and visualized based on selected IDs, columns, and date ranges. The chartOptions computed property dynamically generates configurations for the graph.
* Zoom Controls: Use dataZoom for precise control over the visible range of data.
* Tooltip and Legend: Enhance usability by providing detailed insights into the data.
* Data Fetching: The fetchData method retrieves data based on user-defined filters and updates the graph dynamically.
* Includes error handling to ensure users are informed of issues during data fetching.
* Data Export: The exportData method supports exporting graphs and data in multiple formats with customizable options, including graph type and export path.

**5. Backend Integration**

* Fetch Data and Export Data: Use axios to communicate with backend APIs, passing necessary parameters like database path, selected columns, date range, and IDs.
* Responses from the backend provide the data needed for visualization and export.

### Column Dropdown Component:

The Column Dropdown component in the IMWEBs Viewer application allows users to interactively select columns for data operations such as visualization, analysis, and export. This component adapts to the currently active page (Project, Table, or Graph) and provides a tailored user interface to manage column selections effectively.

Key Features:

**1. Parameter Selection**

For Project and Table Pages:

* Select Columns: Allows users to select multiple columns for data operations.
* Export Columns: Enables users to select specific columns for exporting data.

For Graph Page:

* X-Axis Selection: Displays a dropdown filtered to allow only columns with date type values.
* Y-Axis Selection: Allows users to select multiple columns excluding the date type column.

**2. Dynamic Updates**

* The component automatically fetches and updates columns when a new table is selected.
* Selected columns and export configurations are synchronized with the Vuex store to maintain a centralized state.

**3. Watchers**

Monitors changes to the selectedDbsTables prop:

* Automatically triggers a Vuex action to fetch columns for the newly selected (db, table) list, ensuring the dropdowns remain in sync with the data.

### Database Dropdown Component:

The Database Dropdown Component is an essential part of the IMWEBs Viewer application, designed to provide hierarchical navigation and selection for databases and tables. It renders a tree-like structure where users can explore and choose a database and its associated tables dynamically. This component ensures seamless data selection and integrates tightly with the Vuex store for centralized state management.

Key Features:

**1. Hierarchical Tree Structure**

Dynamically converts a flat list of databases into a tree structure.

Nodes are classified as:

* Folder Nodes: Represent databases.
* Table Nodes: Represent tables under the selected database.

**2. Interactive Node Selection**

Users can select:

* Databases: Triggers an update to the selected database and fetches associated tables.
* Tables: Updates the selected table for further operations such as data analysis or visualization.

**3. Dynamic Data Updates**

Automatically updates the tree structure when new databases or tables are fetched:

* Database Updates: Rebuilds the tree when a new database list is retrieved.
* Table Updates: Adds table nodes to the appropriate database dynamically after fetching.

**4. Implementation Details**

Tree Data Conversion

* Converts a flat array of database paths into a nested tree structure.
* Handles platform-specific path separators (/ for Unix, \ for Windows).
* Ensures unique IDs for each node and differentiates between folders (databases) and tables.

Adding Tables Dynamically

* Locates the selected database node in the tree using the findNode method.
* Clears existing table nodes under the database to prevent duplication.
* Appends new table nodes to the database node's children.

Node Selection

* Database Selection: Updates selectedDbsTables and triggers the fetchTables action to retrieve tables for the selected database.
* Table Selection: Updates selectedDbsTables for further operations.
* Databases Watcher: Rebuilds the tree when the databases list changes.
* Tables Watcher: Dynamically updates the tree when tables are fetched for the selected database.

**5. Usage Flow**

* Initial Rendering: On mounting, the component fetches the list of databases and converts it into a tree structure using the listToTree method.
* Database Selection: Selecting a database triggers an update to selectedDbsTables and fetches associatedbles.
* Table Selection: Selecting a table updates selectedDbsTables for subsequent operations.

### Export Configuration Component:

The Export Configuration Component provides users with an intuitive interface to configure export options for data and graphs. It integrates seamlessly with the Vuex store for state management, ensuring consistency across the application.

Key Features:

**1. Export Path Configuration**

* Allows users to specify the file path where the exported file will be saved.
* Input is sanitized using DOMPurify to prevent potential security risks.

**2. Filename Configuration**

* Enables users to define the filename for the exported file.
* Updates dynamically as the user changes the input, ensuring real-time updates to the Vuex state.

**3. Export Format Selection**

Users can choose the desired export format from a dropdown menu.

Supported formats include:

* Data Exports: CSV, Text
* Graph Exports (conditionally displayed on the Graph page):
* Excel (with embedded graph)
* Image formats: PNG, JPG, JPEG, SVG, PDF

**4. Graph Type Selection**

On the Graph page, users can specify the type of graph to export:

* Updates are synchronized with the Vuex store for consistent graph settings.

**5. Usage Scenarios**

Configuring Data Exports:

* Specify export path and filename.
* Select CSV or Text format for exporting table data.

Configuring Graph Exports:

* Choose a graph type (Bar, Line, or Scatter, Line & Scatter, Scatter & Bar, Bar & Line, and Bar, Line & Scatter).
* Select an appropriate file format (e.g., PNG, Excel, PDF) for saving the graph visualization.

### Export Table and Stats Component:

The Export Table and Stats Component provides users with a flexible interface to choose whether to export data tables, statistical summaries, or both. It uses a multiselect dropdown to offer intuitive options, conditionally presenting choices based on the application state.

Key Features:

**1. Dynamic Option Filtering**

Conditionally displays export options:

* Table: Always available for selection.
* Stats: Shown only if the conditions for statistics (based on selectedStatistics and selectedMethod) are met.
* Ensures users are presented with relevant options tailored to their current configuration.

**2. Multiselect Dropdown**

Utilizes the vue-multiselect library for a feature-rich dropdown interface:

* Multiple Selection: Users can select one or both options (Table and Stats).
* Tagging: Selected options are displayed as tags for better visibility.
* Dynamic Styling: Adjusts tag sizes based on the number of selected options.

**3. Usage Flow**

* Initial Render: The component determines available export options based on the current selectedStatistics and selectedMethod.
* User Interaction: Users select one or both options (Table and Stats) from the dropdown.
* State Update: The selected options are stored in selectedOptions and synchronized with the Vuex exportOptions state.

### Folder Tree Component:

The Folder Tree Component provides a hierarchical view of directories and files, allowing users to navigate and interact with nodes dynamically. It supports recursive structures, node toggling, and custom behavior based on the active page in the IMWEBs Viewer application.

Key Features:

**1. Dynamic Hierarchical Tree**

Tree Structure:

* Represents a nested list of folders (folder type), files (file type), and database or table nodes (database and table types).
* Uses recursive rendering for child nodes, enabling unlimited depth.

Node Icons:

* Displays contextual icons for each type of node:
* Folder: 📁 (collapsed) or 📂 (expanded)
* Database: 🗄️
* File/Table: 📄

**2. Node Interaction**

Toggling Nodes:

* Expands or collapses folder nodes (folder or database) on click.
* Emits a select event to inform parent components of the selected node.

Automatic Expansion:

* Automatically expands specific nodes based on the active page:
* Expands database nodes for specific pages.
* Expands model-related nodes (Model01\Output\Scenario\_2) for the specific pages.

**3. Implementation Details**

Utilizes recursive rendering for child nodes:

* A folder-tree component instance is created for each node with children, allowing the tree to handle deep nesting seamlessly.

Dynamic Node Interaction

* Toggles the expanded state of folder or database nodes.
* Emits a select event to notify the parent component of the selected node.
* Propagates the select event from child components to the root, enabling interaction tracking at all levels.

Automatic Expansion

* Expands specific nodes based on the active page:
* Expands database nodes for the Table page.
* Expands model-related nodes for pages.
* Searches for nodes by name or ID and expands them, including all child nodes.

**4. Usage Flow**

* Tree Initialization: Renders the tree structure based on the treeData prop.
* Node Interaction: Users can click nodes to expand or collapse them, triggering the toggleNode method.
* Event Propagation: The select event propagates from child nodes to the parent, enabling centralized node selection handling.
* Automatic Expansion: The tree adjusts its expansion state based on the page prop whenever it changes.

### Interval Dropdown Component:

The Interval Dropdown Component allows users to select and configure data aggregation intervals for both application usage and export operations. It ensures that the selected and export intervals are synchronized while enforcing logical constraints on the relationships between these intervals.

Key Features:

**1. Dual Interval Configuration**

Select Interval:

* Determines the data aggregation level for the current analysis or visualization.
* Options: Daily, Monthly, Yearly, Seasonally.

Export Interval:

* Configures the interval for exported data, which can either match or exceed the selected interval.

**2. Interval Validation**

* Ensures that the export interval is not less granular than the selected interval.
* Provides user feedback with an alert if the selected export interval is invalid.
* If the export interval is less granular than the selected interval, an alert is displayed, and the exportInterval is reverted to match the selectedInterval.
* Triggered when the export interval changes. Validates the new export interval and updates the Vuex state if valid.

**3. Usage Flow**

* User Interaction: Users select an interval from the Select Interval dropdown. Optionally, they configure a different interval for export in the Export Interval dropdown.
* Validation: The system validates that the export interval is equal to or greater than the selected interval.
* State Synchronization: Updates are stored in Vuex and propagated across the application.

### Message Box Component:

The Message Box Component provides a dynamic and user-friendly interface for displaying notifications and alerts to users. Integrated with Vuex, it ensures a centralized and consistent mechanism for managing application messages, enhancing user feedback and interaction.

Key Features

**1. Dynamic Message Display**

Messages are rendered dynamically from the Vuex messages state.

Supports multiple message types for contextual feedback:

* Info: General information (info).
* Success: Success notifications (success).
* Warning: Warnings about potential issues (warning).
* Error: Error messages (error).

**2. Smooth Animations**

* Uses Vue's transition-group for smooth fade-in and fade-out animations when messages are added or removed.

**3. User Dismissal**

* Each message includes a close (✕) button, allowing users to dismiss individual messages manually.
* Dismissal triggers the sliceMessage Vuex action, which updates the state.

**4. Positioning and Layout**

* Positioned at the bottom-right of the screen for unobtrusive visibility.
* Displays a stack of messages with a maximum width for readability.

**5. Implementation Details**

Message Types

* The message-box element dynamically applies a class based on the message type, enabling distinct styling for each type of message (info, success, warning, error).

Event Handling

* The close button triggers the sliceMessage method, removing the specific message by its index in the messages array.

**6. Usage Flow**

* Message Addition: Messages are added to the Vuex messages state via the pushMessage action. New messages appear dynamically in the message box.
* Message Dismissal: Users can dismiss a message by clicking the close button. The sliceMessage action removes the message from the Vuex state.
* Automatic Removal: Messages can be configured to disappear after a set duration using a timer in the Vuex actions.

### Selection Component:

The Selection Component provides a comprehensive interface for configuring and managing data selection and export parameters. It integrates tightly with Vuex for centralized state management and dynamically adjusts its behavior based on the current data type and user input.

Key Features:

**1. Model Metadata Display**

Read-Only Fields:

* Displays key model information such as start and end dates, and the aggregation interval (daily, monthly, etc.).
* Read-only fields ensure that these values are immutable by the user but clearly visible for reference.

**2. Dynamic ID Selection**

Supports selection for both in-app analysis and export configurations:

* Uses vue-multiselect to provide a searchable, multi-select dropdown for selecting data IDs.
* Selected IDs: Affects the application's current data view.
* Export IDs: Configures which IDs are included in the exported data.

**3. Flexible Date Range Selection**

Supports multiple date formats (date and text) depending on the date type:

* Time Type: Renders a calendar picker for precise date selection.
* Month Type: Provides a text input for entering month values.

**4. Export Configuration**

* Enables users to set independent date ranges and IDs for export operations.
* Synchronizes with the Vuex state to ensure consistent export settings.

**5. Dynamic Interval Capitalization**

* Automatically capitalizes the model interval for display in a user-friendly format.

**6. Implementation Details**

Dynamic Behavior

* The daType property determines whether date or text inputs are rendered for start and end date fields.
* Dynamic handling of date ranges and IDs ensures flexibility and adaptability to different data configurations.

**7. Usage Flow**

* Initial Display: Metadata fields (Model Start Date, Model End Date, Model Interval) are pre-populated with read-only data.
* ID and Date Selection: Users select data IDs and define date ranges for in-app analysis and export configurations.
* Vuex Synchronization: Changes to selections are immediately reflected in Vuex, ensuring consistency across the application.

### Statistics Dropdown Component:

The Statistics Dropdown Component provides a user-friendly interface for selecting statistical operations and data conversion methods within the IMWEBs Viewer application. It leverages the vue-multiselect library for advanced dropdown functionality, ensuring flexibility and ease of use.

Key Features:

**1. Method Conversion Selection**

Allows users to select one or multiple methods for data conversion.

Options include:

* Equal, Average, Sum, Maximum, Minimum.

**2. Statistics Selection**

Enables users to choose statistical operations for data analysis.

Available options:

* None, Average, Sum, Maximum, Minimum, Standard Deviation.

**3. Multi-Select Functionality**

Utilizes the vue-multiselect library to support:

* Multiple selections.
* Tags for each selected item.
* Searchable dropdowns for efficient selection.

**4. Usage Flow**

* Method Selection: Users select one or more methods from the Method Conversion dropdown. Updates the Vuex selectedMethod state.
* Statistics Selection: Users choose statistical operations from the Select Statistics dropdown. Updates the Vuex selectedStatistics state.
* Real-Time Updates: Changes are reflected immediately in the application via Vuex.

### Login:

The login page of the IMWEBs Viewer application provides users with a secure authentication interface to access the system. It is designed with a simple and intuitive user experience while maintaining robust authentication functionality.

Key Features:

**1. User Interface**

* Background Video: A visually appealing background video that plays in a loop to enhance the user experience.
* Login Panel: A centered panel where users can enter their login credentials.
* Site Logo: Placeholder for the site logo, enhancing brand identity.
* Form Fields: Input fields for username and password with validation.

**2. Form Handling**

* Username and Password Input: Users can enter their email address and password to log in.
* Submit Button: A button to submit the login form.
* Error Message: Displays error messages if the login fails.

**3. Authentication**

* Login Method: Handles the login process by sending a POST request to the backend API with the entered credentials.
* Auto Login: When using a Tauri app, it will automatically log in with "default" as the username and password.
* Token Storage: Stores the received authentication token in localStorage for session management.
* Error Handling: Displays appropriate error messages if the login fails.

**4. Styling**

* Responsive Design: Ensures the login page is responsive and looks good on all devices.
* Custom Styles: Includes custom styles for the login panel, form fields, and buttons to match the application's theme.
* Styling Author: The styles were authored by Dr. Michael Yu.

### Map:

The Map page in the IMWEBs Viewer application provides users with a comprehensive interface for visualizing geospatial data. It integrates advanced mapping capabilities using Leaflet, making it a core component for spatial data analysis and presentation.

Key Features:

**1. Folder and Column Navigation**

* Database Dropdown: Allows users to select the database containing the geospatial data.
* Column Dropdown: Enables selection of specific columns from the table for visualization.

**2. Settings Panel**

* Selection: Enables filtering of data based on specific IDs or criteria.
* Interval Dropdown: Sets data aggregation intervals (e.g., daily, monthly, yearly).
* Statistics Dropdown: Configures statistical metrics for the data.
* Export Configuration: Sets parameters for exporting maps and associated data.
* Fetch Map: Loads the map data to be visualized.
* Export Map: Exports map data in user-specified formats.

**3. Map Display**

* Dynamic Mapping: A map is rendered using Leaflet with customizable options.
* GeoJSON and Raster Layers: Supports multiple types of geospatial data, including GeoJSON and raster images.
* Tooltips: Display detailed information about map features.
* Legend: Customizable legend to handle multiple datasets.
* Full-Screen Mode: Allows users to view the map in full-screen mode for better visualization.

**4. Functional Highlights**

* Legend Customization: Legends dynamically adjust to show color levels and associated data ranges.
* Multi-Layer Support: Supports multiple layers of geospatial data, including polygons, lines, and points.
* Data Lookup: Efficiently maps each feature ID to its corresponding properties for rendering.
* Interactive Features: Includes click and hover events to display detailed information and statistics.
* North Arrow: Adds a north arrow to the map for orientation.
* Error Handling: Ensures users are informed of issues during data fetching and exporting.

**5. Backend Integration**

* Fetch Data and Export Data: Uses axios to communicate with backend APIs, passing necessary parameters like database path, selected columns, date range, and IDs.
* Responses from the backend provide the data needed for visualization and export.

### Calculator Component:

The Calculator Component in the IMWEBs Viewer application provides a versatile interface for performing mathematical operations on selected columns. It supports both manual and automatic modes, enabling users to create and preview formulas dynamically.

Key Features:

**1. Mode Selection**

* Auto Mode: Automatically generates formulas by combining a user-defined number with selected columns using a chosen operator.
* Manual Mode: Allows users to manually construct formulas by selecting columns, numbers, and operators.

**2. Formula Preview**

* Displays a real-time preview of the formula being constructed.
* Updates dynamically as users interact with the component.

**3. Column Selection**

* Lists all available columns, excluding geospatial and metadata columns (e.g., dateType, idColumn).
* Users can select columns to include in the formula.

**4. Calculator Grid**

* Provides a grid of buttons for numbers, operators (`+`, `-`, `\*`, `/`), and special actions (`.` for decimal, `⌫` for backspace, `=` for saving the formula).

**5. Formula Saving**

* Saves the constructed formula to the Vuex store (`mathFormula` state).
* Displays a success message upon saving.

**6. Real-Time Updates**

* Changes to the formula are reflected immediately in the application via Vuex.

**8. Popup Design:**

* Displays as a centered modal with a semi-transparent background overlay.
* Includes smooth fade-in animations for enhanced user experience.

**9. Backend Integration:**

* The Calculator Component does not directly interact with the backend but integrates seamlessly with the Vuex store to update the application's state.

This component enhances the analytical capabilities of the IMWEBs Viewer by enabling users to create custom calculations on selected data columns efficiently.

# Security

## Cross-Site Scripting (XSS) and the Importance of Input Sanitization in State Management

Cross-Site Scripting (XSS) attacks exploit vulnerabilities in applications by injecting malicious scripts into web pages, often through user inputs. If an application displays unsanitized user input in the DOM, it can allow unauthorized code execution, leading to severe security risks like data theft, session hijacking, and unauthorized access to sensitive resources.

In our application, Vuex manages critical state data, including user-inputted text fields. Although Vue automatically escapes interpolated text, unsanitized input can be rendered through methods such as v-html or passed to other components that may inadvertently expose the data. Thus, even when user input is stored temporarily or solely in state management libraries (such as Vuex), sanitizing input at the state update level is essential for three key reasons:

Future DOM Rendering: Vuex state often serves as a data source for computed properties or component props that render content in the DOM. Sanitizing inputs before updating Vuex state ensures that any future rendering is secure and cannot lead to XSS vulnerabilities.

Data Persistence: Vuex state can be saved to local storage or shared across components and services. If this data is unsanitized, it increases the risk of XSS if any part of the application later renders it without strict sanitation.

Enhanced Reusability and Security: Sanitizing at the point of state update creates a global layer of security, reducing the risk of introducing XSS vulnerabilities as the application scales or is refactored.

For this application, implementing a sanitization library such as DOMPurify when updating Vuex state is an effective approach. This strategy protects the app by ensuring text data is sanitized at entry, regardless of where or how it might be used or rendered later. By adopting this practice, we reduce the likelihood of XSS vulnerabilities and build a more resilient security model within the state management lifecycle.

# Migration from Electron to Tauri

Initial Setup

Electron was chosen for its ability to package web technologies into desktop applications, but issues with high memory usage, high installation time and large package sizes led to the decision to switch to Tauri. Migrating back to Electron is possible with very little time.

## Challenges & Solutions

Rewriting the package managers (package.json, tauri.conf.json) for communication between the Vue frontend and Python backend in Tauri.

Ensure the Python backend runs as a Tauri sidecar with minimal Rust setup, allowing execution of Python .exe or .py files.

Handling window events to minimize command prompts when launching Python services.

The above challenges were solved using the Tauri documentation.

## Performance Gains

**The following metrics were measured only for the completed Project page of the app:**

Memory Usage: Tauri reduced the application's memory footprint by over 96%, from approximately 107 MB to 3.5 MB, compared to Electron.

Installation Time: The .msi file for Tauri required admin access, allowing it to complete installation in under 1 minute. The .exe file for Tauri took slightly longer, about 1 minute, primarily due to copying a 1.06 GB test model. The Electron .exe file required more time, completing in under 3 minutes, as it involved copying the 1.06 GB test model and running Chromium.

Build Size: Tauri reduced the final build size from 3.99 GB (Electron) to approximately 1.19 GB, of which 1.06 GB is the test model.

Startup Time: The application startup time decreased by 35% in Tauri compared to Electron.

## Disadvantages of Tauri

Tauri is relatively newer compared to Electron, so its ecosystem, tooling, and community support are still evolving, though it is rapidly growing.

Since Tauri relies on the system's webview, it may not support the latest web standards as quickly as Electron, which uses an updated Chromium engine.

## Conclusion

The IMWEBs Viewer desktop application successfully transitioned from Electron to Tauri, yielding significant performance improvements in memory usage, build size, and responsiveness. While Tauri presented initial integration challenges, its lightweight nature and performance benefits outweighed the complexities.

# GitHub Actions for Release Bundles

## Overview

To streamline the release process for the IMWEBs Viewer desktop application, GitHub Actions was configured to build release bundles for Linux, Windows, and macOS. The primary targets were:

* **Linux**: ‘.deb’ and ‘.rpm’ packages
* **Windows**: ‘.nsis’ and ‘.msi’ installers
* **macOS**: ‘.app’ and ‘.dmg’ packages

This CI/CD pipeline ensures that the application can be consistently built and distributed across platforms with minimal manual intervention.

## Workflow File Setup

The GitHub Actions workflow file defines the build process:

1. Environment Setup:

* Install Tauri dependencies, including Rust and Node.js.
* Configure platform-specific build tools (e.g., ‘dpkg’ for ‘.deb’, ‘rpmbuild’ for ‘.rpm’, ‘makensis’ for ‘.nsis’).
* Set up Python for backend compatibility.

2. Building Release Artifacts:

* Use the Tauri CLI to build platform-specific packages.
* Ensure that external binaries (like the Python backend) are bundled correctly.

3. Release Creation:

* Upload build artifacts to the GitHub release page as part of the pipeline.

## Challenges

Setting up the workflow involved several challenges:

1. Cross-Platform Tooling:

* Ensuring all required tools were available across GitHub-hosted runners for Linux, macOS, and Windows.
* Resolving dependency conflicts between Node.js and Rust versions on different platforms.

2. Python Backend Packaging:

* Bundling the Python backend as a sidecar required ensuring that ‘apppy.exe’ or equivalent files were correctly included for all platforms.

3. AppImage Error (Linux):

* While ‘.deb’ and ‘.rpm’ packages were successfully built, creating an AppImage consistently failed due to unresolved issues related to dependencies.
* The error indicated missing runtime libraries, even when explicitly included in the build environment.
* After extensive troubleshooting that took a significant amount of time, the issue was finally resolved by copying the hashed dependencies directly into /usr/lib/ folder.
* This fix allowed the AppImage builds to proceed successfully, and they were reintroduced into the release pipeline.

4. Environment Variables for Testing:

* During testing on Windows, assets failed to load due to missing .env files, which were only available locally.
* The solution involved uploading .env variables as GitHub Secrets and writing them to a .env file during the CI/CD job run.

5. Signing and Notarization:

* For macOS `.dmg` and Windows `.msi` installers, configuring code-signing certificates and notarization processes. These processes require platform-specific adaptations and involve additional costs for acquiring certificates.
* Due to these costs, code-signing and notarization have not been considered at this time.

6. Artifact Size Optimization:

* Ensuring that bundled artifacts included only necessary resources to minimize download size.

## Workflow Highlights

Below is a link to the snippet of the GitHub Actions workflow configuration:

<https://github.com/shahviransh/ECCC-IMWEBs-Viewer/blob/main/.github/workflows/main.yml>

### Current Status

The GitHub Actions pipeline successfully builds and uploads release bundles for:

* Linux: ‘.deb’, ‘.rpm’, ‘.AppImage’
* Windows: ‘.nsis’, ‘.msi’
* macOS: ‘.app’, ‘.dmg’

## Conclusion

The integration of GitHub Actions for release bundles significantly improved the reliability and efficiency of the distribution process.

# Seamless Integration of Web Application and Tauri

## Introduction

Integrating a web application with a Tauri-based desktop sidecar executable can significantly enhance functionality and user experience. However, ensuring seamless interaction without conflicts between the two environments poses challenges, especially in managing file uploads and handling initial setup configurations.

## Avoiding Conflicts Between Web App and Tauri

When developing a system that operates both as a web application and a Tauri desktop application, the following considerations are crucial to avoid conflicts:

* API Endpoint Differentiation: Since the Tauri application interacts with the backend differently than a web-based frontend, conditional logic should be implemented to distinguish between web requests and local desktop interactions.
* Dependency Handling: The web and Tauri applications may require different dependencies. It is essential to structure the project so that dependencies do not interfere with each other.
* Security Concerns: The Tauri application may have access to local system resources, whereas the web app runs in a sandboxed browser environment. Ensuring that permissions are appropriately handled can prevent security vulnerabilities.

## File Upload Management

Handling file uploads separately for both environments is critical since the web app and Tauri interact with the backend differently.

### Web Application

* Uses standard HTTPS requests to upload files to the backend.
* Relies on frontend frameworks (such as Vue.js) to manage user file selections and uploads via a proxy API endpoint.
* The backend (Flask) processes and saves the uploaded files.

### Tauri Application

* Can directly access the filesystem for file operations as the app runs locally.
* Uses Rust bindings to interact with the OS, allowing for efficient local file handling.
* Instead of sending files through the web API, it can directly send files path to backend.

## Connecting to IIS Web Server on Windows

For deployments where the frontend is hosted on an IIS (Internet Information Services) web server in Windows. This setup requires:

* Frontend with HTTPS: The web frontend (served by IIS) can use HTTPS for secure communication with clients over the internet. Since the frontend is exposed to the internet, it must handle HTTPS to protect the data being transmitted.
* Backend with HTTP (Local): The backend (Flask app) will be deployed behind IIS, and since it is on the same machine as the frontend, it doesn't necessarily need to be exposed to HTTPS externally. The communication between the frontend (IIS) and backend (Flask) on the same machine can use HTTP. IIS will act as a reverse proxy, forwarding requests from the frontend (via HTTPS) to the Flask backend (over HTTP).
* SSL/TLS Management by IIS: Since the SSL/TLS encryption is managed by IIS, the web frontend can securely communicate with IIS over HTTPS.
* Proxy API Setup:
  + We can configure URL Rewrite and ARR (Application Request Routing) in IIS to forward requests to the Flask backend. The backend will be running on HTTP, so requests to the /api endpoint in the frontend can be forwarded internally to the backend.
  + The web app will use HTTPS to communicate with the IIS frontend, and IIS will forward these requests to the backend over HTTP (locally).

## Instructions for Setting Up and Running the Application on Windows Server

1. Install Git:

* Download the Git installer from https://git-scm.com/downloads.
* Run the installer and follow the setup instructions.

2. Install Miniconda:

* Download the Miniconda installer for Windows from https://www.anaconda.com/docs/getting-started/miniconda/install.

Run the installer and follow the setup instructions.

3. Install Node.js and npm:

* Download the Node.js installer from https://nodejs.org/en/download.
* Run the installer and ensure npm is installed alongside Node.js.

4. Clone the Repository:

* Open a terminal and run:

git clone https://github.com/shahviransh/ECCC-IMWEBs-Viewer.git

cd ECCC-IMWEBs-Viewer

5. Create and Configure the Conda Environment:

* Create a virtual environment:

conda create -n venv python=3.12

* Install backend dependencies:

conda install -n venv -c conda-forge gdal geopandas pyogrio -y

conda run -n venv pip install --no-cache-dir -r backend/requirements.txt

6. Update .env File:

* Open the .env file in the https://github.com/shahviransh/ECCC-IMWEBs-Viewer folder.
* Set PRODUCTION=True.
* Set VITE\_API\_BASE\_URL=http://webgis.csahs.uoguelph.ca/IMWEBs-Viewer

7. Start the Flask Backend with Waitress:

* Run the following command to start the Flask server in production mode in background at http://127.0.0.1:5000:

Start-Process -NoNewWindow -FilePath "conda" -ArgumentList "run -n venv python backend/apppy.py"

8. Build the Frontend:

* Run the following command to build the frontend. Use the path to dist folder to load the website.

npm install

npm run build

9. Configure Reverse Proxy:

* Set up a reverse proxy to route requests: Map http://webgis.csahs.uoguelph.ca/IMWEBs-Viewer/api to http://127.0.0.1:5000/api.

## Automating Updates with AutoUpdate.ps1

To ensure the latest code is fetched from GitHub, the backend and frontend are rebuilt, and the website is updated periodically, follow these steps:

1. Edit AutoUpdate.ps1:

* Open the AutoUpdate.ps1 script in a text editor.
* Locate the $projectDir variable and update it with the absolute path to the ECCC-IMWEBs-Viewer directory. For example: $projectDir = "C:\ECCC-IMWEBs-Viewer"

2. Schedule AutoUpdate.ps1 to Run Periodically:

* Open Task Scheduler in Windows.
* Click Create Task.
* Set a name like IMWEBs Auto Update.
* Select “Run whether user is logged on or not”
* Choose a trigger (e.g., Daily or Weekly at a preferred time).
* Choose Start a Program as the action.
* Set powershell.exe as the program/script.
* In the Add arguments field, enter:
* -ExecutionPolicy Bypass -File "C:\ECCC-IMWEBs-Viewer\AutoUpdate.ps1"
* Click Finish to save the task.

3. Schedule MoveToIIS.ps1 to Run Periodically:

* Open Task Scheduler in Windows.
* Since only moving files to the IIS folder requires administrative privileges, we create a separate script to follow the principle of least privilege.
* Click Create Task.
* Set a name like IMWEBs Move to IIS.
* Select "Run whether user is logged on or not".
* Check "Run with highest privileges" (required for admin operations).
* Choose a trigger (e.g., Daily or Weekly at a preferred time).
* Choose Start a Program as the action.
* Set powershell.exe as the program/script.
* In the Add arguments field, enter:
* -ExecutionPolicy Bypass -File "C:\ECCC-IMWEBs-Viewer\MoveToIIS.ps1"
* Click Finish to save the task.

This will ensure that the latest code is fetched, the backend and frontend are rebuilt, and the website is updated automatically based on the defined schedule.

## Challenges in Initial Setup

Setting up a project that supports both a web app and a Tauri application comes with its own set of challenges:

* Environment Configuration: Setting up development and production environments to support both web and Tauri builds without conflicts.
* Backend Conflict: Ensuring that builds for both applications remain consistent and do not introduce unexpected behavior.
* Platform-Specific Issues: Differences in behavior across Windows, macOS, and Linux need to be accounted for, particularly in how file handling and permissions work.

## Conclusion

By addressing these challenges and maintaining a clear separation of concerns, integrating a web application with Tauri can be both seamless and efficient. Proper handling of API interactions, model folder uploads via proxy API endpoints, and secure communication with an IIS-managed frontend ensures a smooth user experience across both platforms.

# Discussion

## Challenges Encountered & Solutions

**1. Electron Issues**

* High memory consumption and difficulty managing native system resources led to performance bottlenecks.
* **Solution:** Migrated to Tauri for better resource efficiency and system integration.

**2. Tauri Migration**

* Challenges in managing Python as a sidecar process and ensuring application window-backend synchronization.
* **Solution:** Utilized Rust commands and Tauri’s on\_window\_event to control Flask lifecycle and process synchronization.

**3. Database Queries**

* Fetching all data at once caused performance issues.
* **Solution:** Optimized SQLite queries using lazy-loading techniques to fetch only necessary entries.

**4. State Management**

* Complex component communication required a structured state approach.
* **Solution:** Implemented Vuex for global state handling and efficient inter-component communication.

**5. Backend Termination**

* Flask backend continued running after the Tauri app was closed, causing resource conflicts.
* **Solution:** Used on\_window\_event in Tauri along with a Rust command to ensure Flask shutdown on app close.

**6. Browser vs. Tauri Height Difference**

* Layout gaps appeared due to differences in height rendering between browser and Tauri.
* **Solution:** Applied window.isTauri to dynamically adjust height based on platform.

**7. Path Management**

* Inconsistent file path handling across Windows and Linux.
* **Solution:** Standardized path formatting using forward slashes (/) to improve cross-platform compatibility.

**8. Map Export Issues**

* Exporting map images led to misaligned or missing GeoJSON layers, incorrect tile and vector layer rendering.
* **Solution:** Switched to dom-to-image-more for improved vector layer capture and accurate rendering.

**9. Running Python Sidecar in Tauri**

* Release packages installed with admin privileges required Python to execute as admin as python edits and creates files in an admin location.
* **Solution:** The backend directory is determined dynamically using platformdirs.user\_data\_dir, which provides a writeable location appropriate for storing application data per user. A TempFiles folder is created within this directory (e.g., AppData\Local\IMWEBs-Viewer\TempFiles on Windows). As a result, the Python server can start and operate normally, storing and modifying files without requiring elevated permissions.

**10. Shapefile Reprojection and GeoJSON Handling**

* Some shapefiles lacked a specified spatial reference system (SRS), requiring a default. Processing large shapefiles in-memory led to performance issues. Existing GeoJSON data needed efficient updates without redundant reprocessing.
* **Solution:** Set a default SRS (EPSG:26917, UTM Zone 17N) for shapefiles without a specified SRS. Implemented batch processing with a buffer size of 1000 features, reducing memory overhead and improving performance. Compared shapefile metadata (feature\_count, extent, field\_names) with existing GeoJSON metadata before writing to avoid unnecessary updates. Used a generator function to efficiently append new features to an existing GeoJSON dataset without excessive memory use.

**11. Shapefile Saving and Zipping with Multithreading**

* Saving large shapefiles (> 100 MB) was taking too long, causing performance bottlenecks.
* Solution: Utilized multithreading with ThreadPoolExecutor to parallelize saving shapefiles. This approach sped up the process by allowing concurrent execution of I/O-bound tasks, reducing the overall time required to handle large datasets.

By addressing these challenges, we improved performance, stability, and cross-platform compatibility in our application.

## Recommendations for Future Work

Feature Expansion: Add support for more data formats and dynamic data queries.

UI Enhancements: Improve the overall user experience by introducing more interactive elements and optimizing performance further.

Cross-Platform Testing: Conduct extensive testing across multiple operating systems to ensure consistent behavior.

# References

Vue.js Documentation: <https://vuejs.org>

Python Documentation: <https://docs.python.org/3.12/>

Python Flask Documentation: <https://flask.palletsprojects.com>

GitHub Documentation: <https://docs.github.com/en>

Electron Documentation: <https://www.electronjs.org/docs>

Tauri Documentation: <https://tauri.app>

Vue-echarts: <https://github.com/ecomfe/vue-echarts>

Miniconda Documentation: <https://www.anaconda.com/docs/getting-started/miniconda/main>

Node Documentation: <https://nodejs.org/docs/latest/api/>

NPM Documentation: <https://docs.npmjs.com/>

Git Documentation: <https://git-scm.com/doc>