Evaluating Chart Libraries for Flexible and Adaptable Data Visualization

Bachelor-Thesis
for the course Medieninformatik
submitted by

Nadine Eunous
student number: 43372

on Wednesday 21st May, 2025

at the Stuttgart Media University
to obtain the academic degree of Bachelor of Science

First Examiner: Prof. Dr. Ansgar Gerlicher
Second Examiner: Dr. Karin Hauff

Ehrenwörtliche Erklärung

Hiermit versichere ich, Nadine Eunous, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Evaluating Chart Libraries for Flexible and Adaptable Data Visualization“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungs- rechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, den 17. Mai 2025
Ort, Datum

Unterschrift

Abstract

Business intelligence has become an established standard in the industry, and the ability to quickly create ad-hoc visualizations is becoming increasingly important. This motivates the investigation into which charting library is best suited for this purpose. Many chart libraries offer quick and easy implementations of basic graphs but might lack the flexibility and ease of use required for more complex, custom-built visualizations. Development teams often face challenges when implementing specific design or data requirements, which can lead to increased development time, reliance on custom components, and potentially inconsistent user experiences. This thesis evaluates how different libraries perform in respect to various relevant predefined criteria and when implementing sample use cases.

This study considers a broad range of popular JavaScript libraries before choosing the most fitting libraries for further examination. Afterwards the three top picks: Chart.js, ApexCharts and ECharts, will be thoroughly evaluated. The evaluation is guided by criteria such as ease of use, responsiveness, community and support, bundle size and integration flexibility. To evaluate the chart libraries three use cases with specific requirements were defined and created with all three libraries. The sample charts include a trend line diagram, a Pareto diagram and a Sankey chart.

This comparison study should help developers decide which JavaScript charting library best suits their needs by combining hands-on implementation with analytical evaluation. The results aim to offer insights into some of the strengths and limitations of these libraries.

Kurzfassung

Business Intelligence hat sich in der Industrie mittlerweile als Standard etabliert, und die Fähigkeit, Ad-hoc-Visualisierungen schnell zu erstellen, wird immer wichtiger. Dies motiviert die Untersuchung, welche Charting-Library sich hierfür am besten eignet. Viele Charting-Libraries bieten schnelle und einfache Implementierungen für grundlegende Diagramme, weisen jedoch häufig nicht die Flexibilität und Benutzerfreundlichkeit auf, die für komplexere, individuell gestaltete Visualisierungen erforderlich sind. Entwicklungsteams stehen oft vor Herausforderungen, wenn es darum geht, spezifische Design- oder Datenanforderungen umzusetzen. Dies kann zu einem erhöhten Entwicklungsaufwand, einer stärkeren Abhängigkeit von benutzerdefinierten Komponenten und potenziell inkonsistenten Benutzererfahrungen führen. Die Bachelorarbeit untersucht, wie verschiedene Libraries in Bezug auf vordefinierte Kriterien und bei der Umsetzung von Beispielanwendungsfällen abschneiden.

Diese Studie berücksichtigt eine breite Auswahl populärer JavaScript-Charting-Libraries, bevor die am besten geeigneten Libraries für eine vertiefte Untersuchung ausgewählt werden. Anschließend werden die drei zur weiteren Analyse ausgewählten Libraries – Chart.js, ApexCharts und ECharts – eingehend evaluiert. Die Bewertung orientiert sich an Kriterien wie Benutzerfreundlichkeit, Community und Support, Responsiveness, Bundle-Size und Flexibilität bei der Integration. Zur Evaluierung der Charting Libraries wurden drei Anwendungsfälle mit spezifischen Anforderungen definiert und mit allen drei Libraries umgesetzt. Die Beispiel-Diagramme umfassen ein Trendlinien-Diagramm, ein Pareto-Diagramm und ein Sankey-Diagramm.

Die Vergleichsstudie soll Entwicklern dabei helfen, die JavaScript-Charting-Library zu wählen, die am besten zu ihren Anforderungen passt. Durch die Kombination von praktischer Implementierung und analytischer Bewertung sollen Erkenntnisse über die Stärken und Schwächen der jeweiligen Library gewonnen werden.

Contents

1 Introduction

1.1 Motivation

In today's world, data has become a very valuable asset across industries. Organizations rely on data to drive strategic decision-making, helping them identify trends, optimize resources, and gain competitive advantages. [1] In the context of operational excellence, data is used to improve efficiency, streamline workflows, and eliminate inefficiencies. [2] Similarly, in data-driven development, software teams make informed decisions based on usage patterns and performance metrics rather than intuition.

Beyond traditional analytics, machine learning and artificial intelligence further enhance the value of data. Predictive models and AI-driven insights enable forecasts, automate decision-making, and improve the quality and meaningfulness of data. However, regardless of how advanced the analysis is, raw data alone is often difficult to interpret. To extract actionable insights, effective visualization is crucial. Studies have shown that the human brain can identify an image in as little as 13 milliseconds, highlighting the importance of presenting data visually for quick and intuitive understanding. [3]

While technologies like AI contribute significantly to data interpretation, this thesis explicitly focuses on data visualization, not on the development or evaluation of artificial intelligence methods. This study does not explore how data is analyzed, but how it is best communicated to users through visual means.

The aim is to compare different JavaScript charting libraries to evaluate their capabilities, flexibility, and usability and to determine which library offers the best development experience.

1.2 Objectives and Approach

This thesis aims to systematically evaluate different charting libraries. The research method used was an empirical comparison of the libraries by focusing on key criteria for the comparison. This thesis aims to answer the following research questions:

- Research Question 1: Which charting library best fulfills predefined requirements based on relevant use cases?
- Research Question 2: What are the strengths and limitations of the different charting libraries?

To identify the most suitable JavaScript charting library, this thesis follows a structured evaluation process. The first step involves a review of available JavaScript charting libraries, considering a broad range of options. Based on predefined selection criteria—such as popularity, documentation quality, feature set, and community support—a preselection of the most promising libraries will be made.

Following the initial selection, the libraries will be subjected to a detailed assessment based on the criteria ease of use, responsiveness, community support, integration flexibility and bundle size. With the results of a structured evaluation matrix the most fitting charting library will then be chosen.

The preselected libraries will first be examined using a step-by-step approach, beginning with the most basic implementation of the required chart type and progressively incorporating customizations and adaptations to meet specific visualization needs. This process aims to examine the flexibility and adaptability of each library.

To ensure a practical and application-oriented comparison, the evaluation will be carried out in the context of three distinct use cases. These use cases are designed to reflect real-world data visualization needs, covering different complexity levels and requirements. By implementing the same visualization scenarios across multiple libraries, this study aims to provide an objective comparison of their strengths and limitations in a practical development environment. The implementation is available in GitHub [4] and the results can be viewed on the deployed website. [5]

2 Charting Libraries: Landscape and Selection

2.1 Overview of Charting Libraries

Charting libraries have found a place in various programming environments and languages, being integrated into a wide range of applications such as desktop programs, web apps, mobile apps, dashboards for business intelligence, and data science. Due to their diverse application fields, charting libraries have been developed for numerous languages, including C#, C++, Kotlin, Java, Python, and JavaScript. This study will focus on using charting libraries for web applications.

2.2 Benefits of Charting Libraries

There are many reasons why charting libraries are important for data visualization. These pre-made tools can significantly simplify the process of creating visualizations. Developers don't have to piece together the visualizations from scratch, which can be time-consuming and error-prone. The libraries provide you with pre-made visualizations which include all necessary elements such as axes, labels, legends and tooltips —most of which are already tested for usability and accuracy.

Additionally, most charting libraries are designed with ease of use in mind, allowing developers to generate complex visualizations with just a few lines of code. Comprehensive documentation, along with community support, further facilitates the learning process, making them accessible for both beginners and experienced developers.

Another important factor is the integration of UI design principles into prebuilt visualizations. Through hands-on exploration of various libraries, it became evident that

all examined libraries in this thesis incorporate certain design guidelines. For example, fundamental aspects such as the shapes of bars in a bar chart—always rectangular by default—are consistently applied. The Material Design Guide, a design language system developed by Google for mobile devices and web applications, serves as a common reference for best practices in user interface design. Additionally, there are established principles for effective data visualization. While some guidelines are inherently followed by the libraries, others are not strictly enforced by default. However, the examined libraries provide tools and customization options to ensure that charts can be adapted to meet specific design recommendations.[6]

2.3 Preselection of Charting Libraries

2.3.1 Requirements

To identify the most suitable charting libraries for this study, a preselection process was conducted. Since the libraries will be used in the context of a web application, they must be capable of rendering charts efficiently in a browser environment. The selected libraries should meet the following requirements:

- **Open Source Availability** – This means the library must be "[...] distributed with a license that allows access to its source code, free distribution, the creation of derived works, and unrestricted use." [7].
- **High User Base** – A larger user base generally correlates with more community support, making it easier to find resources, examples, and troubleshooting help. The number of GitHub stars was used as an indicator of popularity.
- **Extensive Chart Variety** – The library should support a wide range of chart types to enable diverse data visualizations. At a minimum, it must support the three use cases examined in this study: a line chart, a mixed bar and line graph, and a Sankey chart.
- **Good Documentation** – Clear and comprehensive documentation is essential to simplify the learning process and facilitate quick adoption without requiring extensive familiarization.
- **Ease of Use** – The library should allow for a straightforward implementation and easy flexibility and customization options, enabling users to start creating visualizations with minimal effort but also customize and create more enhanced or complex graphs.
- **Client-Side Execution** – Since the study focuses on chart libraries within a web application, they must be capable of running in the frontend without requiring server-side rendering. This is an important criterion to ensure interactivity and responsiveness. The browser tracks mouse movements and clicks and most frontend charting libraries leverage this data. It also contributes to reducing server load if charts are created on the client.

Another factor which is very important, especially in the context of corporate application development, is data security. This does not pose a risk for any of the reviewed libraries of this study since the data is not sent to a server but processed on the client side. All data processing takes place locally in the browser, meaning that no data is transmitted to third parties.

A potential risk could, however, arise from the use of third-party libraries with their own transitive dependencies, which could contain vulnerabilities. There are tools available that notify of existing vulnerabilities in dependencies for example the built-in command "npm audit" for the package manager NPM, the open source tool Dependabot, the corresponding enterprise solution SonarQube or the security platform Synk. It is also important to use responsive packages which install the new fixed releases as soon as they are published. [8] By using libraries with a large user base and active community support this risk can be mitigated, as these libraries are more likely to be regularly updated and maintained.

By applying these criteria, a shortlist of charting libraries was created for further examination.

2.3.2 Examined Libraries and Selection

Table 1: Criteria for Preselection of Charting Libraries

Library	**Open Source**	**GitHub Stars**	**Charts**	**Good Documentation**	**Client-Side**
D3	✓	110k	several	✓	✓
Chart.js	✓	65.2k	several	✓	✓
Echarts	✓	61.7k	several	✓	✓
Recharts	✓	24.4k	several	✓	✓
Matplotlib	✓	20.1k	several	✓	with pyodide
Plotly	✓	17.2k	several	×	✓
ApexCharts	✓	14.6k	several	✓	✓
Nivo	✓	13.3k	several	×	✓
Victory	✓	11k	limited	✓	✓
MUI X Charts	✓	4.6k	limited	✓	✓

D3

The seemingly most popular library is D3 with 110 thousand GitHub stars.[9] It stands for data-driven documents and facilitates web documents like the Document Object Model (DOM) to create visualizations. With this library visualizations are built piece by piece instead of providing ready-made components. [10] In the case of the basic line graph this meant creating the `<svg>` element, a format for vector based graphics, the axis and axis scales, the graph lines and then grouping them together. While this approach is highly customizable it also requires a lot of effort, specifically a lot of code, to create a simple chart. For this reason this library was not further persued.

Chart.js

The second most popular library is Chart.js with 65.2 thousand GitHub stars.[11] This library uses the HTML5 element `<canvas>` to render its charts, an empty container for graphics which are drawn with JavaScript [12], and offers a variety of ready-made graphs. Chart.js offers a npm-package which provides the charts directly as React components. This library offers many chart types, is easy to use as well as well-documented and was therefore chosen for further evaluation. [13]

ECharts

Next is ECharts with 61.7 thousand GitHub stars [14]. This library gives the user the choice between rendering with `<canvas>` or `<svg>` and offers the highest chart variety out of all libraries, has good documentation and was chosen for evaluation. [15]

Recharts

Recharts has 24.4 thousand GitHub stars and is built directly on React components.[16] This would make it very easy to use in React, on the other hand, its usage is restricted to React applications. It is also built on top of `<svg>` elements and also has a moderate amount of charts. This library was not chosen for full examination because of its restriction to React. It was considered important to use a flexible library which can be used in different frameworks. [17]

Matplotlib and Pyodide

Next a python library was considered in combination with Pyodide, a python distribution for the browser. Pyodide, first released in 2019, is a relatively new package that ports CPython to WebAssembly and Emscripten. It enables the installation and execution of Python packages directly in the browser through micropip.[18] This solution can be very beneficial for developers who are familiar with Python. Especially in the context of Data Science where Python is the predominant language [19], and data visualization has become very important, the data visualization libraries have become very powerful and comprehensive. Some examples of popular Python charting libraries are matplotlib, Seaborn and Bokeh. The library used for testing this was matplotlib. This approach was not chosen for the detailed evaluation since the Pyodide environment initialization takes 4 to 5 seconds causing a short delay in visualization and JavaScript based libraries have faster loading times.[20]

Plotly

Plotly.js has 17.2 thousand GitHub stars and is built with D3.js and stack.gl. [21] Stack.gl is a modular Web Graphics Library (WebGL) ecosystem that emphasizes shader code writ-

ing and provides different tools for creating custom graphics applications using WebGL [22], a JavaScript API for rendering interactive graphics in browsers [23]. There are many charts available, however the documentation could be more extensive, when compared to the other libraries. This library was not chosen mainly due to issues with responsiveness. The documentation mentions a responsive attribute which is used for responsive layouts, however this only responds to a browser window resize, not a resize of HTML elements e.g. the `<div>` element containing the chart.[24]

ApexCharts

ApexCharts has 14.6 thousand GitHub stars [25] and uses `<svg>` to render the charts. More precisely, the SVG.js library is leveraged, a library which simplifies the manipulation of `<svg>` elements, through more concise and intuitive syntax. This library offers many charts and a good well-structured documentation. Additionally, integration into various frameworks like React, Angular and Vue is seamless since there are wrapper components available for each environment. This library was chosen for further evaluation. [26]

Nivo

Nivo is next with 13.3 thousand GitHub stars. [27] This library is built on top of React and D3 and offers many charts. While there is extensive documentation available, the absence of a comprehensive guide for getting started can make it challenging for new users to get started and become familiar with the library. This, combined with limited flexibility for integration across different frameworks, led to the decision not to pursue further evaluation of Nivo in this study. [28]

Victory

Victory has 11 thousand GitHub stars and is built on top of React, utilizing `<svg>` for rendering [29]. It offers good documentation and a moderate selection of chart types. A notable feature of Victory is its ability to extend seamlessly to both Android and iOS platforms while maintaining an identical API. This ensures a consistent user interface across different operating systems and enhances the developer experience by providing a unified development process. However, Victory was not selected for this study due to the absence of a Sankey chart—one of the key chart types evaluated—and its restriction to React and React Native applications [30]

MUI X-Charts

MUI X-Charts is the last considered library with 4.6 thousand GitHub stars.[31] It is part of the Material UI library used for integrating ready-made React UI components which incorporate the Material Design guidelines made by Google. According to its official website, it leverages "[...] D3.js for data manipulation and SVG for rendering".[32] This

can make it a good option for developers already using MUI, since this makes the UI- as well as the development experience consistent. However, this library also restricts usage to React applications and another limiting factor is, that the charting components are a relatively new feature for the MUI library, with the first stable release of the charts package, v6.18.0, dating to November 3. 2023. Many chart types, for example the Sankey chart, Gantt chart and Funnel chart, are still in development. For these reasons MUI will not be evaluated in this study.[31]

2.3.3 Canvas vs. SVG Rendering

During the selection process it became evident that there were two main rendering types that charting libraries used: canvas and SVG. Canvas renders by drawing pixels while SVG renders vector-based.[[33]] The key advantages of SVG rendering are that the graphics are scalable while maintaining a high quality as well as the manipulation with the Document Object Model which allows for easy interaction and flexibility. Its main disadvantage is that the bigger the graphics with larger numbers of elements the slower the rendering becomes. This is, on the other hand the key advantage of canvas rendering. Its disadvantage of not having built-in interactivity, is not relevant since the charting libraries are developed to include interactive charts. For these event handlers had to be implemented by developers. Another disadvantage is, on updates the entire graph has to be drawn a new, whereas SVG can access individual elements to update them. [34] Since the chart applications of this study do not require very large datasets the better performance of canvas was not viewed as a deciding factor to disqualify SVG libraries and vice versa for the canvas libraries interactions are already integrated, so this also does not pose a problem in our study.

Selection

The initial research and preliminary practical testing led to the selection of Chart.js, ECharts, and ApexCharts for further evaluation.

3 Evaluation Framework

All diagrams created in this thesis are to be used in the context of managing agile teams within the SAFe framework, which stands for Scaled Agile Framework. It consists of a set of organizational and workflow patterns for implementing and scaling agile practices across enterprises, as well as defining different roles and responsibilities—all aiming to promote collaboration and efficiency [35].

SAFe also introduces structured organizational constructs such as Agile Release Trains (ARTs) and Solution Trains. ARTs are multiple agile teams that work together to deliver value on a regular basis, [36] while Solution Trains coordinate multiple ARTs to deliver large-scale solutions. [37]

To ensure these value streams function effectively, SAFe defines several flow metrics. Flow refers to the smooth, uninterrupted, and fast movement of work through the value stream and is measured using various indicators.[38] One of these metrics is Flow Load, which evaluates whether there is a balance between demand and capacity. It helps identify if teams are overloaded with too many parallel tasks, which can slow down progress. Another key metric is Flow Predictability, which assesses how successfully teams, ARTs, and Solution Trains deliver business value relative to their planned objectives. This metric serves as an indicator of reliability and planning accuracy within the organization.[39]

3.1 Definition of the Use Cases

To effectively examine the charting libraries, three different use cases were chosen. The example use cases were carefully chosen to include a diverse range of chart types, allowing for a more comprehensive assessment of each library's capabilities in handling different data visualization scenarios. Each graph is implemented to meet specific criteria so that the adaptability of the libraries can be evaluated. The examples were carefully drafted to include data visualization guidelines from the previously mentioned Material Design Guidelines so that the incorporation or adaptability to best practices in readability, accessibility, and usability are also considered in the evaluation. For each example a sketch was created to visualize the ideal expected outcome. The following section will introduce the three use cases and their respective sketches.

3.1.1 Line Chart

The first example displays whether a monthly target was reached or not. The graph used for this visualization is a line chart, which consists of the target line and the actual line. The target line is dashed, and the actual line is red or green, depending on whether it meets the target. The y-axis represents the number of completed tickets, while the x-axis shows the months. If the number of tickets exceeds the target line, the actual line is displayed in green; otherwise, it is red. However, this logic can also be reversed. For instance, if we were visualizing the number of open bug tickets, points exceeding the target line would appear red, as the goal in this case is to minimize the number of bugs.
To improve accessibility and align with Material Design guidelines—specifically, the principle of not relying on color alone to convey meaning—a second target line can also be incorporated as a visual cue. This additional reference reinforces whether the values at any given point are meeting expectations, making the information more accessible to all users. Furthermore, the different data types were accentuated by using the varying line textures. [6] It was decided to only include horizontal grid lines as the vertical lines do not provide any benefit for a better readability and only overcrowd the chart.

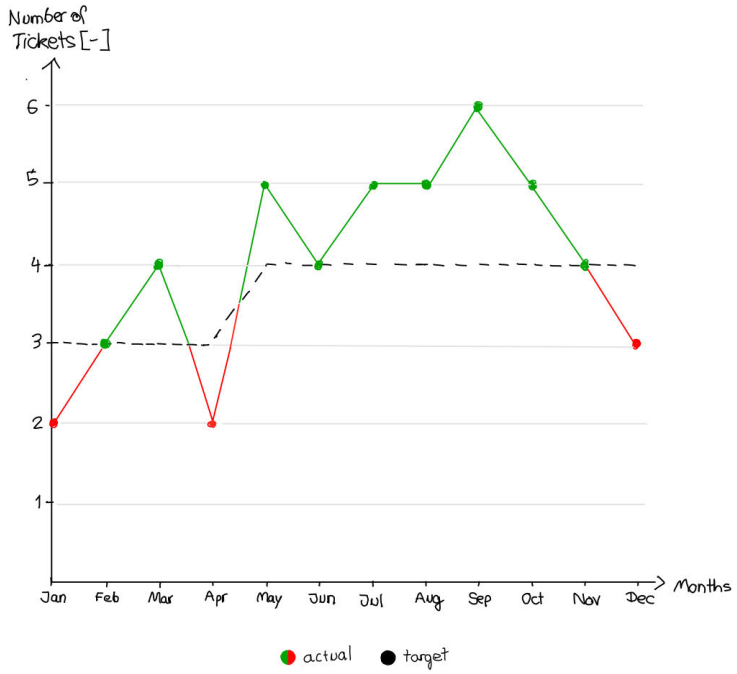

Figure 1: Sketch of the Line Chart

3.1.2 Pareto Chart

The next use case is a Pareto diagram, which is utilized in the context of Hoshin Kanri, a management methodology. The core principle of Kanri is to help companies establish a clear direction by defining a few key objectives while ensuring their successful implementation by aligning them with operational activities. To support this process, the method offers various tools, one of which is the Pareto diagram. Pareto analysis is designed to help identify the most significant causes or levers for a problem based on available data. This technique, rooted in the Pareto principle (also known as the 80/20 rule), suggests that roughly 80% of the effects come from 20% of the causes. By focusing on these critical few causes, organizations can prioritize their efforts to achieve the greatest impact. The diagram is a mixed chart consisting of a line and bars. The bars represent a type of category, they could be different segments or different products, in my example I used the fictitious car model classes, A-, B-, C-and D-Class. The line chart tracks the cumulative percent. A vertical splitting line as well as a horizontal line is added at 80% to visualize which bars are still within the 80%. In the example visualization the number of open tickets is the decisive metric. [40]
Design guides considered in this example were to not overload the graph with too many axis labels, for this reason the scales of the axis were adapted. The chart baseline starts at 0, to keep a correct perception of the data.

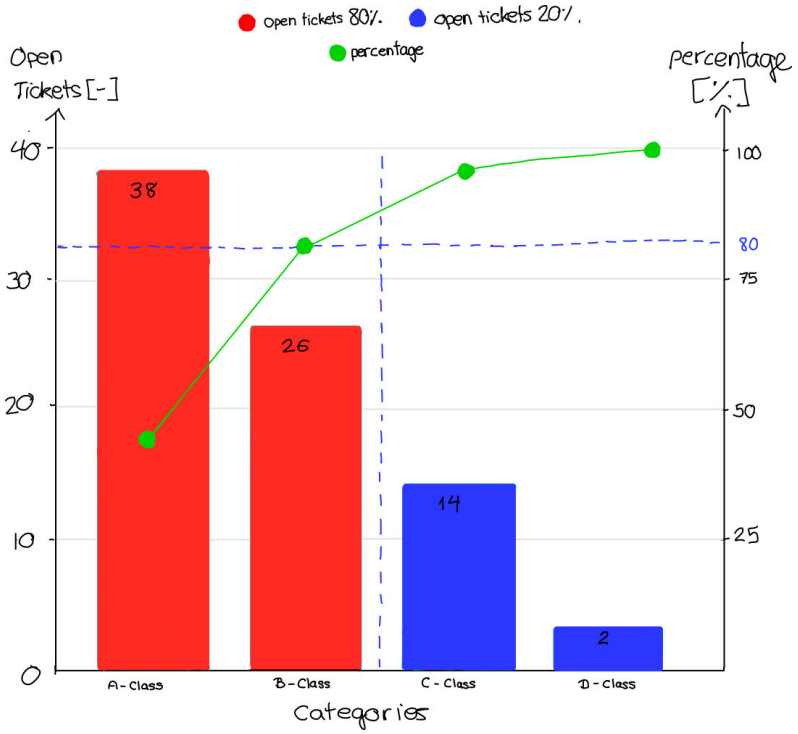

Figure 2: Sketch of the Pareto Chart

3.1.3 Sankey Chart

The last example is a Sankey chart. This type of chart is used to visualize the flow of data and show transitions between different states. The chart consists of nodes and arcs, the nodes are the states and the arcs are the changes in states since they connect the source nodes to the target nodes. The width of the arcs is proportional to the amount of data flowing through them making it easier to identify the most significant transitions. [41] The example visualization shows the flow of tickets through different states, such as backlog, in progress, in review and done, over the course of several sprints. The width of the arcs is proportional to the number of tickets in each state. The chart is used to identify bottlenecks and inefficiencies in the process. Two different versions of this chart will be depicted the first version will be a simpler Sankey chart which shows the flow of all tickets throughout the sprints. In this graph the colors correspond to the states of the tickets. Between different states there are also color transitions within the arcs. The second version will be a continuation of the previous Pareto use case, and will highlight the flow of the tickets which are contained in the crucial 80% of the Pareto chart. Through this visualization, the viewer can get deeper insights into the most impactful areas of the workflow, enabling more targeted improvements and optimizations.

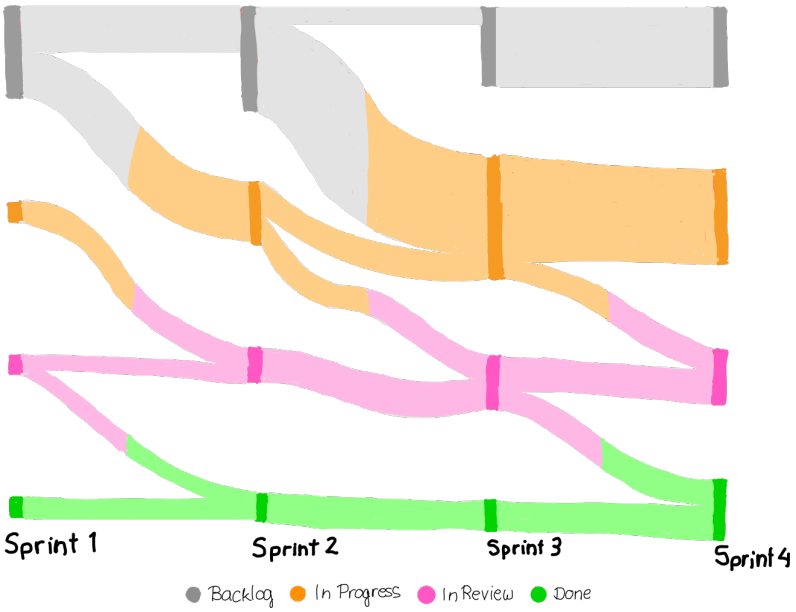

Figure 3: Sketch of the Sankey Chart: Version 1

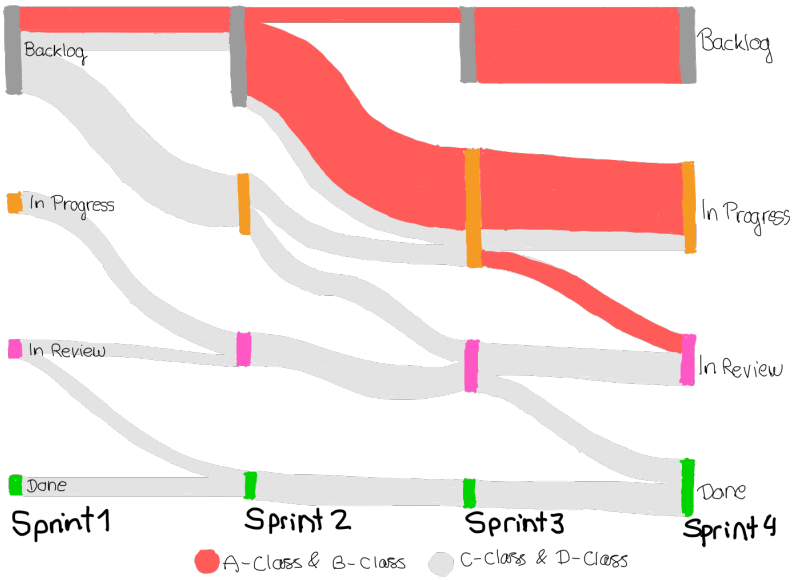

Figure 4: Sketch of the Sankey Chart: Version 2

The material design guide recommends legends for desktop applications be placed under the chart, this was applied in all three use cases. Another characteristic embodied by all graphs but not explicitly mentioned in the guide is that the grid lines are more subtle by using a lighter color, this was also applied in the use cases.

3.1.4 Evaluation Criteria and Matrix

The evaluation of the charting libraries will be based on the following criteria:

Table 2: Evaluation Criteria, Weight Rates, and Scores

Criteria	Weight Rate	Score
Ease of Use	4	1-3
Responsiveness	3	1-3
Community and Support	2	1-3
Integration Flexibility	1	1-3
Bundle Size	1	1-3

A weighted decision matrix will be used to evaluate the charting libraries. All criteria will be rated on a scale from 1 to 3, with 1 being the lowest and 3 being the highest. These scores will then be multiplied by the weight rates to calculate the final score for each library.

The criteria ease of use refers to how easy it is to get started with the library, how intuitive the API is, and how flexible the library is for customization and implementing specific requirements? Can all requirements be implemented? This criterion is crucial and is therefore rated as the highest weight rate of 4.

Responsiveness evaluates how well the library performs in terms of responsive sizing. This is also very important for a good web design and therefore weighted at 3.

Community and support assesses the activity level of the community around the library, how well the library is maintained, and the quality of support available. Since this is always a useful time saving factor this point is weighted at 2.

Integration flexibility considers whether there is a Node Package Manager (NPM) package available and whether the library is also available in other JavaScript frameworks. While this can simplify code and save time, implementation would still be possible if the library was only available in JavaScript, so this receives the lowest weight rate of 1.

Bundle size examines the library's file size and how it handles tree shaking. While this can be very important for big data applications, it is not so crucial for the use cases in this study, so it is weighted at 1.

4 Implementation

4.1 Basics of the Implementation

Chart.js, ApexCharts and ECharts all have React wrappers available which make it possible to seamlessly integrate the charts into a React environment as components so these were leveraged in the implementation. All three libraries have a similar structure for creating charts. The data and the chart options have to be set and passed on to the library through the component props. The data is passed as an array of objects, all other settings and chart adjustments are controlled via the chart options. The chart options object contains all the settings for the chart, such as the type of chart, the legend and tooltips just to name a few.

4.2 Setup

Chart.js

To use the library, first it must be installed via the node package manager NPM. The core library is installed with the terminal command `npm install chart.js` and the React wrapper with `npm install react-chartjs-2`. For importing the library there are two options: either the entire Chart.js library can be imported with an auto package import or each used element can be imported individually as shown in the example. With the auto import the library is imported with automatic registration of all elements, however this means a larger bundle size. The individual import is more efficient in terms of bundle size but requires manual registration of the elements. [42] Only importing the elements that are needed and avoiding unused code is commonly referred to as tree shaking. [43]

ECharts

The ECharts library is installed with `npm install echarts` and the React wrapper with `npm install echarts-for-react`. In order to take advantage of tree shaking to reduce the bundle size, only the core libraries were imported and only the specific components and modules needed. Afterwards, the used modules have to be registered. The ECharts Handbook also offers support on the implementation of an options type in TypeScript which can be beneficial if users know exactly which options are needed, then TypeScript can provide errors which point out if something is missing.[44]

ApexCharts

The ApexCharts library is installed with NPM with the command `npm install apexcharts`, additionally the React Wrapper has to be installed with `npm install react-apexcharts`.

For this library there is no tree-shaking available. With the import `import Chart from "react-apexcharts"` the library is imported and all components are directly accessible.

4.3 Implementation of Use Case 1: Line Chart

4.3.1 Chart.js

A basic implementation of a line chart in Chart.js is available open source. [45] This is how the chart looks like in the example:

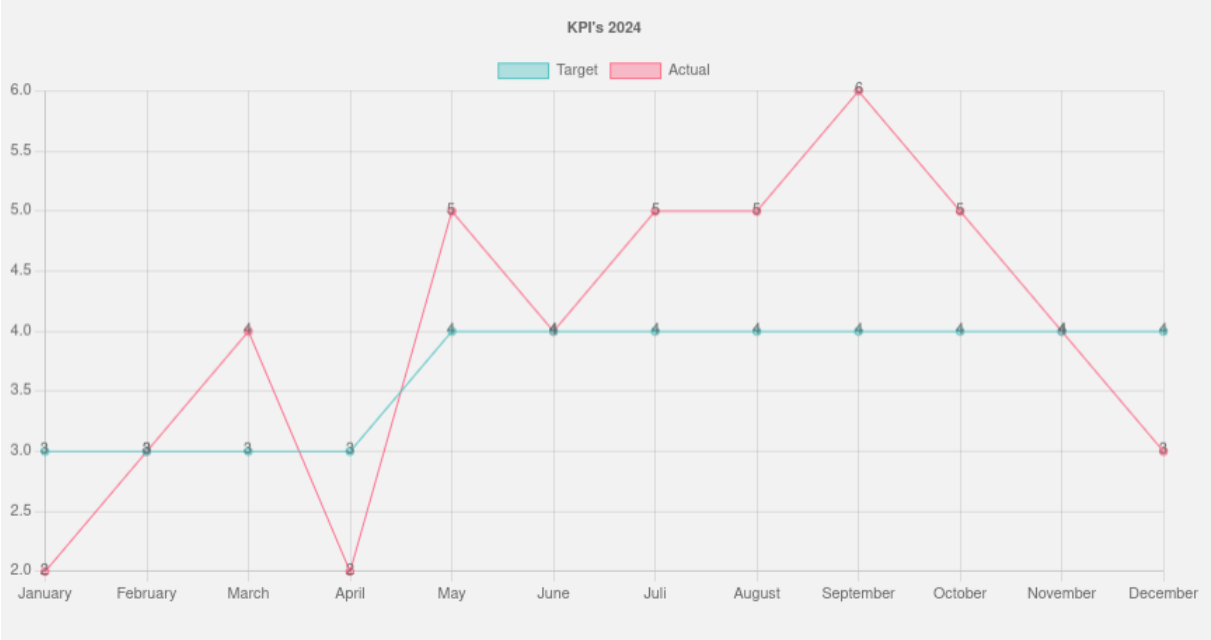

Figure 5: Simple Line Chart in Chart.js

Chart Implementation

The data is loaded in a `useState` which can be beneficial for automatically updating the data on changes, since a re-render is triggered on state changes. While the data is hard-coded in the example and the chart is static for simplicity purposes, for data fetched from an API the data might be dynamic and need regular updates. Within the `chartData` object the labels and the datasets of the chart are set. [46]. The dataset is an array of objects. In the example there are two dataset objects which create the two chart lines for the target and the actual values. Also, settings specific to the dataset are set for instance the `borderWidth` which sets the line width, `label` or the color settings with `backgroundColor` and `borderColor`. In the `options` the universal chart settings are set, for example the title of the chart, the scales of the axes or the settings of grid lines within the chart etc.

Adaption to Use Case

To adapt the line chart to the use case a few changes were necessary. Firstly, the chart legend is automatically placed above the chart. The placement of the legend can be changed within the `options`, so it is underneath. The y-axis scale automatically adapts to the data and the starting value is changed, to revert this change a minimum value of 0 has to be set in the `options`. To label the axes, inside the `options` a `title` must be added for the x- and y-axes. The chart automatically has vertical and horizontal grid lines, the vertical lines can also be removed within `options`. For adapting the line styles the dashed line format of the target line can easily be set within the `dataset`, however the multicolored line proved to be a bit more complex. It took more than one iteration to achieve the desired result. The `segment` option can be use to override the line color. The first approach was to compare each individual actual data point with its corresponding target data point. This meant that the segments would only switch colors at whole values and never between at the exact intersects of the lines like shown in the sketch. While this didn't work for the line segments, the same approach was used for coloring the data points with the `pointBackgroundColor` option which worked well in this case.

In order to achieve the desired line styling it was necessary to change the `scale` of the x-axis to a `linear` scale. By default, the x-axis was set to a `category` scale even without defining the `type`, which meant that no decimal values were accepted as data points. By changing the x-axis to a linear scale also the data passed to the chart had to be adapted. The data array was changed to the TypeScript type `ScatterDataPoint` which is an array of objects containing the points in the format `{x: number, y: number}`. For each section between two points, a check for intersections is done. To calculate the exact intersection points the formula for linear interpolation was used:

$$\mathbf{y}(\mathbf{x}) = \mathbf{y_1} + \frac{\mathbf{y_2} - \mathbf{y_1}}{\mathbf{x_2} - \mathbf{x_1}} \cdot (\mathbf{x} - \mathbf{x_1}) \tag{1}$$

The arrays with the coordinates of the target line and actual line were updated to include all intersection coordinates. Now the line segments can be colored and by including the intersection coordinates the colors change at the exact intersection points. The colors of the actual line are now displays correctly, but there are new markers at the intersection points. The additional markers were removed in the `pointRadius` of the dataset by checking for integer values, for all non integer values the radius was set to 0.

Lastly the chart legend has to be adapted. Since this legend is generated automatically based on the datasets and the defined colors within the datasets, the legend markers only display the predefined colors and only accept a single color by default. In order to show that the actual line is both green and red a third empty and hidden dataset was created. This creates a third label within the legend which can also be labeled "Actual" and shows the second color of the line.

The implementation of this example is available on GitHub.[47] The final result of the line chart in Chart.js can be seen in the following figure:

Figure 6: Final Line Chart in Chart.js

4.3.2 ECharts

The basic implementation of a line chart in ECharts is also available open source. [48] This is how the chart looks like in the example:

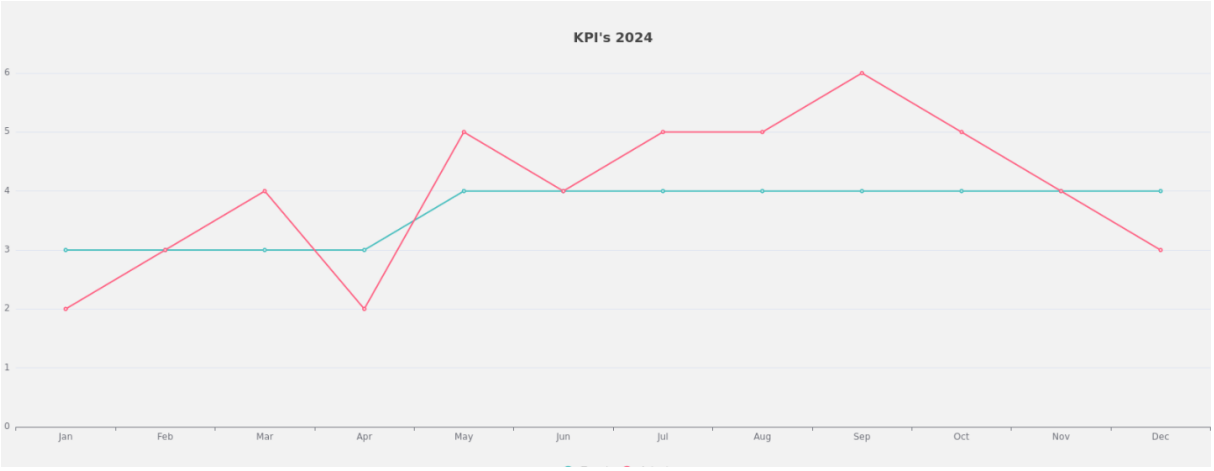

Figure 7: Simple Line Chart in ECharts

Chart Implementation

In the ECharts React component the instance of ECharts which contains only the registered modules is passed as a prop. For this library all chart options and data are passed

together within the `options` prop. Inside the ECharts `options` the data is then set within `series`. Additionally, a size has to be passed for the component via the `style`, because the chart would otherwise not be visible as the size automatically sets to 0 if not set. Otherwise, it follows a very similar structure to Chart.js.

Adaption to Use Case

One of the first noticeable missing elements, is the y-axis line. This has to be explicitly set to be visible within the `yAxis` options. The axis labels are also included within the axis objects and can be set there, but also spacing and margins are not handled automatically and have to be manually included to ensure a good readability and design. As for the line styles, the dashed target line can easily be set within the `linestyle` of the series dataset, for an accurate actual line style a similar approach to Chart.js was used. Both axis have to be of the type `value` in order to accept decimal values and the same calculation of intersections with interpolation was used as in 4.3.1. There is no specific property in this library that can be used for coloring only sections of the line like the `segments` property in Chart.js. However, in a blog post for another library a workaround was introduced which can be used somewhat similarly for ECharts. The blog introduced the idea of using an SVG elements built-in gradient property for applying colors. [49] Within the dataset `lineStyle` attribute a color gradient can be set. This code snippet shows what color settings are needed:

```
color: {
    type: "linear",
    x: 0, y: 0, x2: 1, y2: 0,
    colorStops: colorStops,
    global: false,
},
```

Listing 1: Color configuration in Echarts

With the `linear` type a linear gradient is applied. The next line defines the direction of the gradient. The first two values define the starting point to be in the top left corner and the last two values set the ending point to the top-right corner. The `colorStops` contains an array of objects that define the colors and their corresponding positions in the gradient. Each object has an `offset` with a value between 0 and 1, which specifies the position of the color in the gradient and the color at this position. In the code base the `colorStops` array is dynamically generated based on whether the "Actual" values are above or below the "Target" values. And lastly, with the `global` value we can specify if the gradient is applied globally to the entire chart, then it is set to true, or locally relative to the bounding box of the line, then it is set to false, as is the case in the example. The chart now contains additional marker points at the intersections, the same method was used to hide the excess markers as in Chart.js within the `symbolSize` of the dataset. Also, for the legend the same problem of each legend item only accepting one color occurs. The same workaround, as previously mentioned in section 4.3.1, was used in order to have both red and green visible within the legend.

The full implementation is available on GitHub.[50] The final result is visible in the following figure:

Figure 8: Final Line Chart in ECharts

4.3.3 ApexCharts

A basic implementation of a line chart in ApexCharts can be viewed on the online repository. [51] The figure shows the resulting chart:

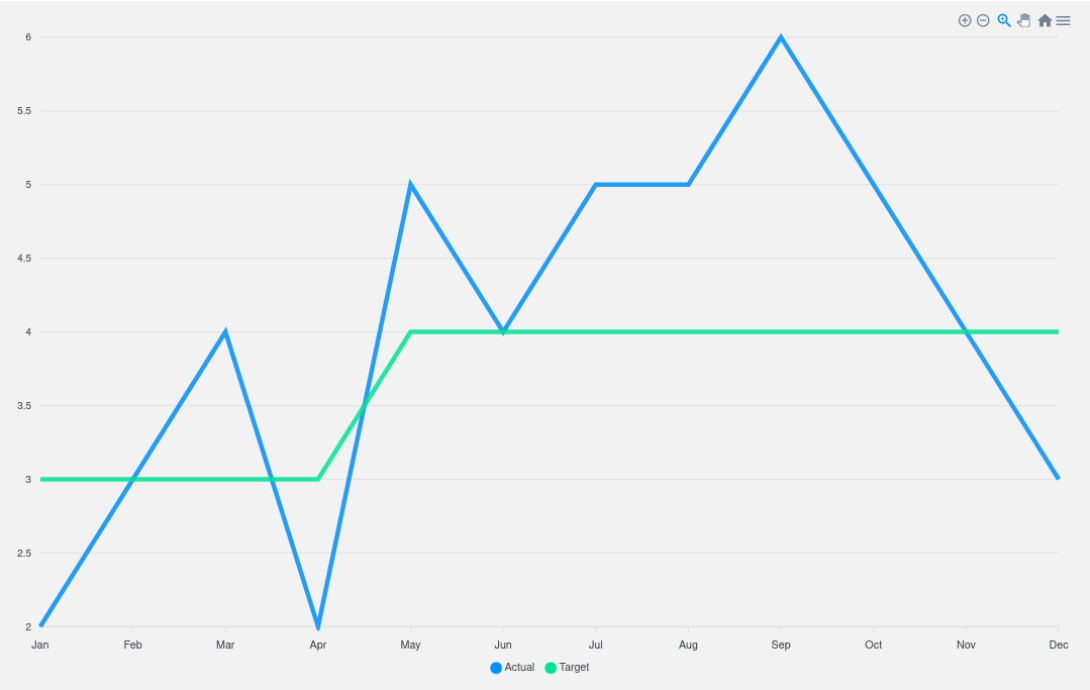

Figure 9: Simple Line Chart in ApexCharts

Chart Implementation

The basic implementation is very similar to the other libraries. The chart component has three props: the `type` with which the graph type is defined, the `data` where the datasets are passed and `options` which contains all other chart options. The minimum value of the x-axis is automatically adjusted to the content, this needs to be reset to 0. The same problems are faced in regard to adjusting the color of the "Actual". This means the x-axis needs to be set to the type `numeric` and the data is changed, so each point is a coordinate in the Cartesian system. No property is available which simplifies styling line sections. The solution for creating multicolored lines introduced in the blog post for Recharts was again used as inspiration. [49] In contrast to the other libraries where the colors are set within the dataset, ApexCharts has a `fill` property within the `options` which can be used to define different fills. These are the settings for the `fill` that lead to the desired outcome:

```
fill: {
    type: ["gradient", "solid"],
    colors: ["#000000"],
    gradient: {
        type: "horizontal",
        colorStops: colorStops.flatMap((stop) => [
            { offset: stop.offset, color: stop.color},
            { offset: stop.nextOffset, color: stop.nextColor},
        ]),
    },
},
```

Listing 2: Color configuration in ApexCharts

The `type` is an array of the fill types, which are `gradient` for the actual line and `solid` for the target line. The gradient need to be horizontal and the `colorStops` are set the same way as for the ECharts library, mentioned in section 4.3.2.

A side effect of changing the x-axis to a numeric scale is that the ticks with the month labels are no longer aligned correctly with the graph data. To resolve this, the `tickAmount` property can be set to `dataJson.length - 1`, ensuring that the number of ticks matches the number of data points. This adjustment aligns the labels correctly with the data points and prevents extra or misaligned ticks. Hiding the additional markers proved to be a little more challenging than in the other libraries. The global `marker` property within the chart options can be used for this purpose as shown below:

```
markers: {
    size: 0.001,
    discrete: coloredData
      .map((point, index) => ({
        seriesIndex: 0,
        dataPointIndex: index,
        size: Number.isInteger(point.x) ? 5 : 0,
      }))
    .filter((d) => d.size > 0),
}
```

Listing 3: Marker Customization in ApexCharts

Firstly, by setting the marker size to 0 all markers are hidden by default. The `discrete` property within `marker` allows customization of the markers and their visibility. The data is passed as an array and each object contains the `seriesIndex` and `dataPointIndex` which are needed to identify the markers. The size of the markers is set to 0 for all non-integer values and 5 for integer values. This now displays all markers correctly however the tooltip animation breaks in the process. It seems the discrete property is not fully compatible with the tooltip animation, the tooltip no longer follows the cursor but stays on the right side of the chart for all markers. Similar issues regarding discrete markers where reported in the ApexCharts GitHub repository already. [52]

The resulting chart is shown in the following figure and the full implementation is available on GitHub.[53]

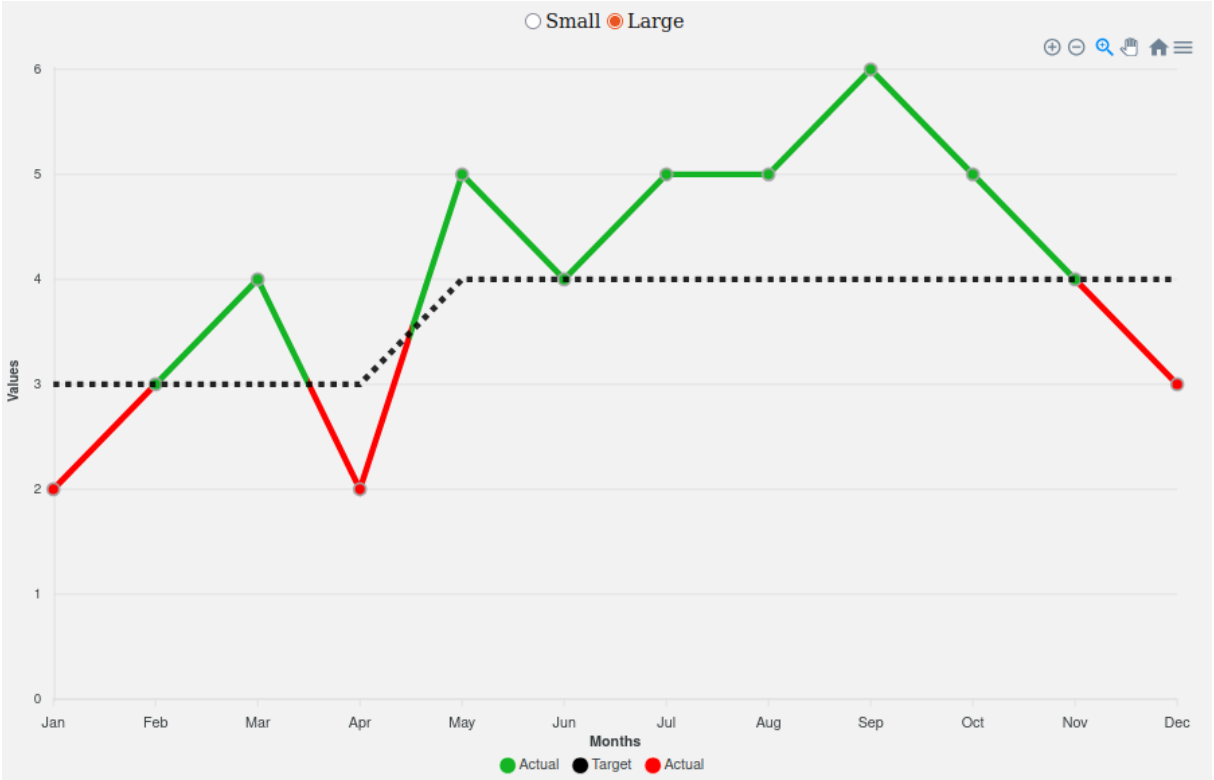

Figure 10: Final Line Chart in ApexCharts

4.3.4 Evaluation

In terms of user interface adjustments, Chart.js required only a few standard modifications, such as repositioning the legend from its default position at the top to the bottom, hiding vertical grid lines, and setting the y-axis to start at zero. Among the three libraries evaluated, Chart.js emerged as the clear winner for the given use case. The key advantage was its built-in segment property, which allowed for a straightforward implementation of differently colored line segments. Moreover, the library enabled easy customization of markers, including conditional styling based on data values, thereby offering a high degree of flexibility with minimal effort.

ECharts, required very little modifications to the standard visual settings. The y-axis line is not visible by default and must be explicitly enabled. Additionally, spacing and margins for axis labels are not automatically managed, necessitating manual adjustment, which is not necessary for the other libraries since they handle this automatically. While ECharts supports extensive customization through its linear gradient color options, which is leveraged for styling the individual line sections, this approach is more labor-intensive compared to the seamless functionality offered by the segment property in Chart.js.

ApexCharts also required minimal user interface adjustments, such as setting the y-axis starting point to zero and enabling the axis line manually. One notable difference is that colors are not defined within individual datasets but rather through a global fill option, which was initially confusing. Similar to ECharts, the implementation of multicolored line

sections relies on a gradient fill approach. While this method allows for highly flexible customization, it demands considerable configuration effort. Marker customization proved to be particularly challenging in ApexCharts. The `markers.discrete` property first needs to identify individual markers to be targeted and styled by specifying the `seriesIndex` and `dataPointIndex`, this approach is less intuitive, as it is defined globally rather than within the dataset. Its usage also introduced unwanted side effects, notably breaking the tooltip animation and thereby negatively affecting user experience.

All three libraries demonstrated limitations in terms of legend customization. Legends are generated automatically and tightly coupled to the datasets, which restricts flexibility. A common workaround involving a hidden dataset was applied across all three to overcome this limitation.

In summary, a fully functioning implementation of the line chart was achieved by Chart.js and ECharts but Chart.js offered the most efficient and developer-friendly experience, particularly for this use case.

4.4 Implementation of Use Case 2: Pareto Chart

The Pareto chart is the next use case. It is a mixed chart which includes a line- and bar-chart. Each chart type needs its own axis, for this reason two y-axes are necessary. For the bar chart different colors are applied to direct the users' focus to the most important information. Additionally, a vertical and horizontal line are included in the chart to mark the 80 percent mark and highlight which bars, more precisely for the example, which car classes, are included in the 80 percent. For this use case it was not necessary to apply a two-step approach, since the example is not so complex and the customization options are not so advanced.

4.4.1 Chart.js

Creating a mixed chart type is possible by adding the `type` property in each dataset individually and setting it to the desired chart type. With the property `yAxisID` also used in the datasets users can create IDs for the y-axes. Within the `scales` object these IDs can be referenced, and the axes can be customized. A few data transformation operations are performed in advance to create the desired data format for the graph, for example, the values for the number of open tickets in each group have to be summed up, and the bars have to be sorted from highest counts to lowest. The data of the percentage line chart is dependent on the values of the bar chart. To achieve this a function was written to use the bar counts and calculate the cumulative percentage values. With each new index they are summed up until the last index where the value reaches 100 percent. Further chart customizations can be done with `plugins`. The object `annotations` contains the property `verticalLine` and `horizontalLine`. Here the `scaleID` to which the annotation is applied is passed as well as the `value` which is the position of the line. The horizontal line was set to 80 percent and the value of the vertical line was set to the index where 80 percent is reached, since the line should not be directly on the bar but between bars, 0.5 was added to the value. Additional customization was applied here, like making the line

dashed and applying a different width and color. Finally, to create the custom bar colors where the bars which contain the first 80 percent are red and the rest of the bars are blue a map function was used to iterate over the data and conditionally set the color of each bar depending on if it is below or above the index where 80 percent is reached.

Lastly, the legend has to be fixed, as the same problem as in the line chart occurs. The legend is generated automatically based on the datasets and the defined colors within the datasets. The legend markers only display the predefined colors and only accept a single color by default. In order to show that the open tickets are visualized both with the red and blue bars a third empty and hidden dataset was created. This creates a third label within the legend which can also be labeled "Open Tickets" and shows the second color of the bars. It is important to create an empty line chart and not a bar chart, because for the bar chart space is automatically reserved for the dataset even if it is empty. This approach for customizing the legend was used for all three libraries.

The final result of the Pareto chart in Chart.js can be seen in the following figure and the full implementation is available online.[54]

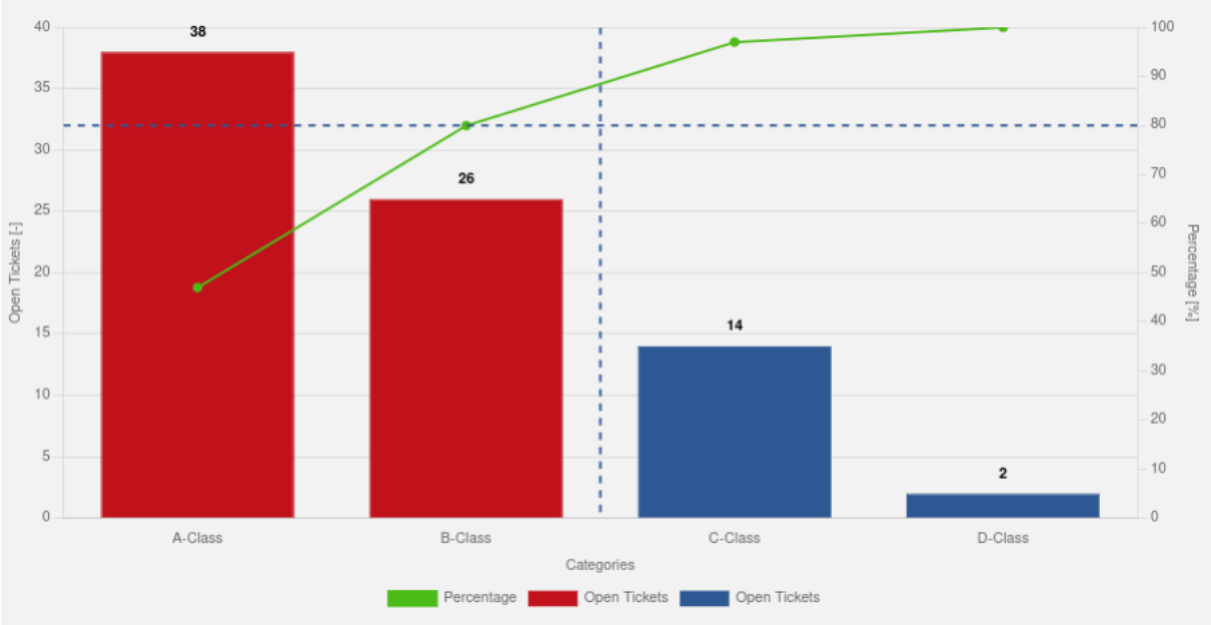

Figure 11: Final Pareto Chart in Chart.js

4.4.2 ECharts

Similarly to Chart.js the mixed bar and line chart can be created by setting the `type` property in the dataset. Again, since multiple datasets are being used, they each need their own y-axis, which can be referenced in the datasets with `yAxisIndex`. The data transformations mentioned in the previous section are also necessary here. Custom coloring based on the index of the data is also easily done, the same function for mapping the colors to the bars was used but in contrast to Chart.js the `itemStyle` property doesn't accept an array of colors but only a single color or, a function to dynamically set each

bar color as applied in the Pareto chart.

The vertical and horizontal separating lines are set within the `series.markLine` object. Setting the horizontal line was unproblematic. Within `markline.data` an array of objects is accepted, so a single or multiple lines can be created. With `yAxisIndex` the y-axis on which this line should be applied is referenced, and the value is passed in within `yAxis`. Additional customization for styling or animation among other things can be given, the API provides a good overview of all customization options. [55].

Adding a vertical line had to be solved differently to Chart.js. The desired outcome was a line used as a separator between the bars that make up the 80 percent and the remaining 20 percent. However inserting a line at decimal values in the bar chart is not supported in the library because ECharts aligns axis markers to discrete category indices rather than continuous or fractional positions, which limits precise placement between bars in a categorical axis setup. To resolve this the `xAxis` had to be converted to `type: ''value''` so the x-axis is treated as a continuous numeric scale. Additionally, it had to be offset by `+ 1`. This shifts the bars so the first bar at index 0 starts at x = 1, this placement is necessary, so the bar doesn't overlap with the y-axis. For the same reason the maximum x-axis value had to be increased by `+ 1` within `xAxis.max`. Since the `category` axis is no longer used, ECharts doesn't automatically show category labels. The `formatter` property is used to set labels manually at the correct positions. After applying all these changes the `markLine` can now be placed at decimal values. The last necessary adjustment is to set the marker value at `xAxis:index80 + 1.5` instead of `+ 0.5` since the first bar is now at the x-value 1. However, in the implementation for the calculation the indices still need to start at 0.

The implementation can be viewed on GitHub and the following figure shows the resulting chart. [56]

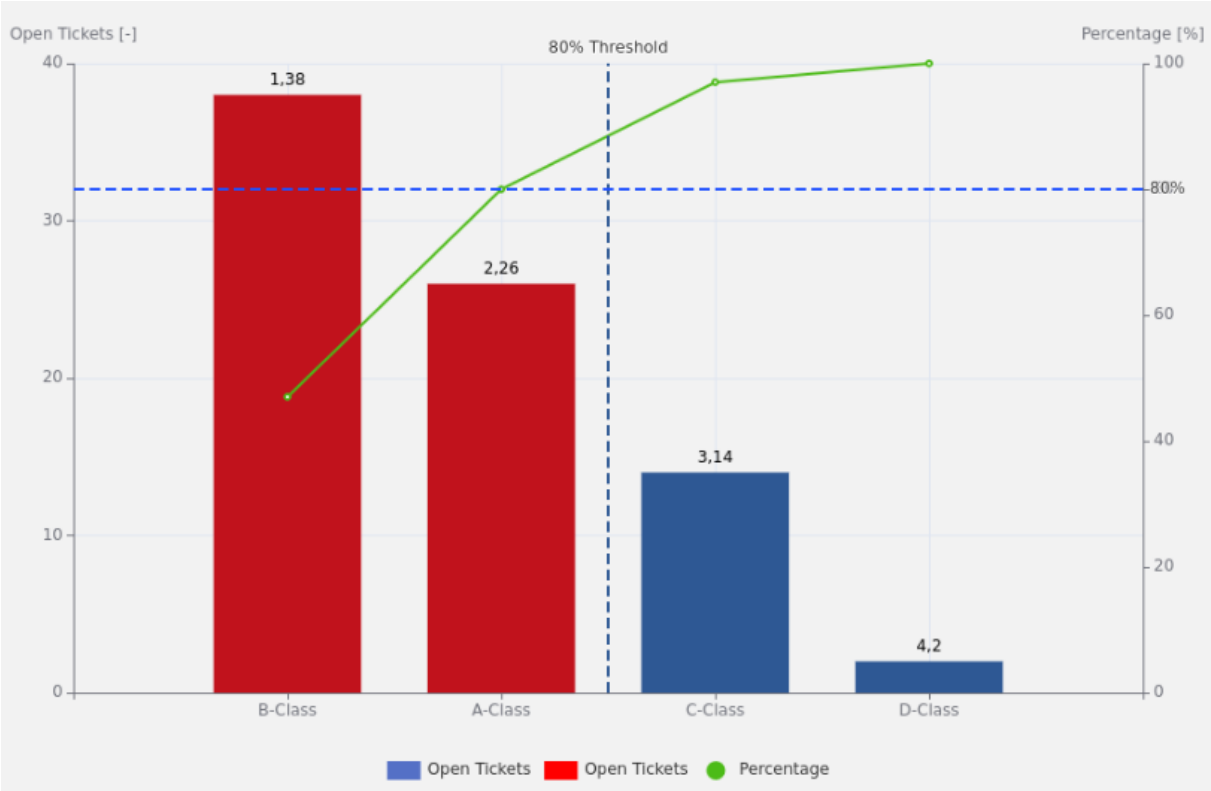

Figure 12: Final Pareto Chart in Echarts

4.4.3 ApexCharts

Similarly to the other libraries different chart types can be combined by adding the relevant `type` within the `series`. The data is transformed beforehand as described in section 4.4.1. Dynamic coloring of the bars can be achieved in the same way as in ECharts, in `series.color`, a function is used to set the color for each index individually. Multiple y-axes can be defined within `yAxis`. By default the first series item is mapped to the first y-axis entry and the second series item is mapped to the second y-axis entry and so on, so the axis doesn't necessarily have to be referenced. To set vertical and horizontal lines the `annotations.xaxis` and `annotations.yaxis` is used. For the horizontal mark line at 80 percent, simply setting the value in the `y` property and referencing the axis for which to apply this line with `yAxisIndex` is sufficient. Further customization options can be found in the documentation [57]. For the vertical line the same issue arises as in ECharts, where the line cannot be placed at decimal values between the bars. The same approach of converting the x-axis values to `type:"numeric"` was used. The actual bar indices are implicitly positioned at 1, 2, 3, ..., n, within the `formatter` property. Lastly, since the bars are positioned starting from 1 instead of the index value 0 also the annotation's `x` has to be changed to `index80 +1`.

In our optimal solution sketch the vertical line spans across the entire graph for better readability, however in the current implementation the line only spans from the first to last bar. Another issue is that the hover tooltip is not handling the conditional coloring and defaults to black because it can't resolve a fixed color. While there is possibly a

solution for this and some promising approaches were found [58], they involve more effort and complexity than the other libraries and were not pursued further.

The final result of the Pareto chart in ApexCharts can be seen in the following figure and the full implementation is available on GitHub.[59]

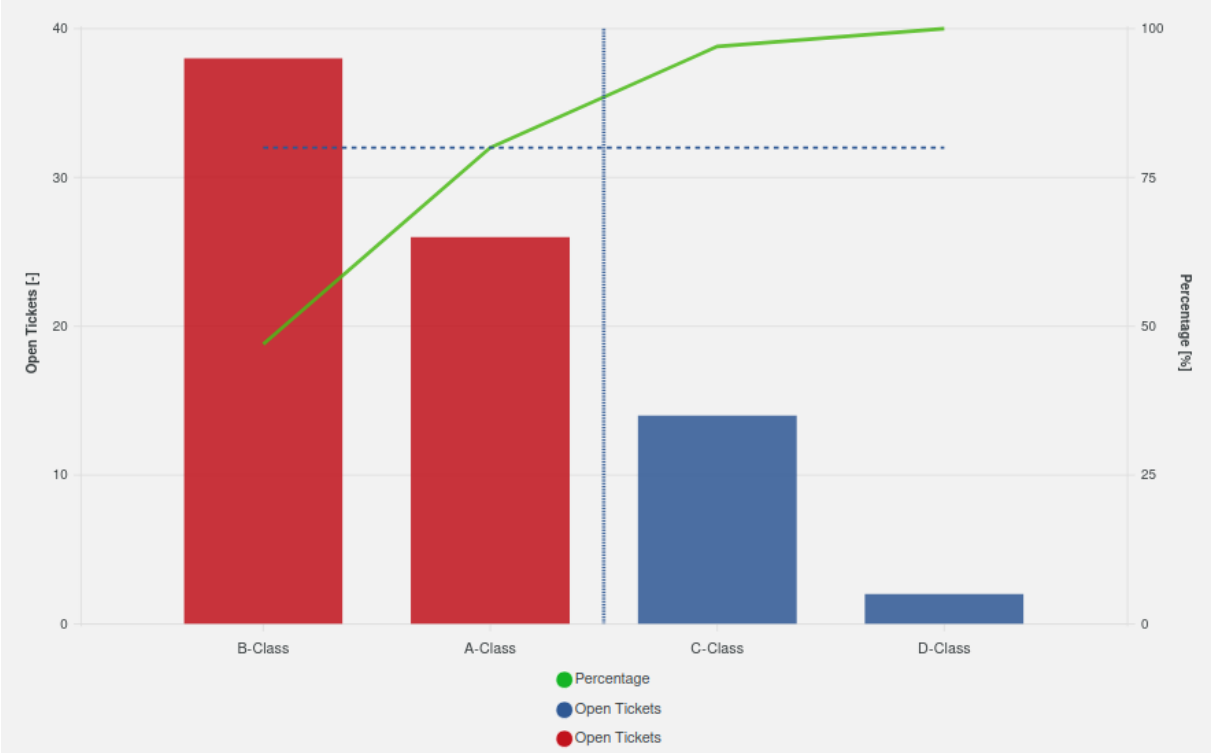

Figure 13: Final Pareto Chart in ApexCharts

4.4.4 Evaluation

For this use case, both Chart.js and ECharts achieved the desired outcome. The only UI-related improvement needed for ApexCharts was extending the line annotation to span the entire width of the graph. However, the ApexCharts implementation still exhibited issues with the hover tooltip, as it did not display the correct colors.

Chart.js offered the most convenient solution overall. Its default categorical x-axis supports placing chart annotations at decimal values, allowing annotations to appear between bars without extra configuration. In contrast, achieving this behavior in ECharts and ApexCharts required switching the x-axis to a numeric scale and manually formatting the tick labels to display the category names.

Overall, Chart.js is provided the most suitable tooling for this use case due to its straightforward implementation and bug-free behavior. ECharts ranks second for also providing a functional solution with no perceivable issues. ApexCharts comes last, as not all bugs could be resolved easily.

4.5 Implementation of Use Case 3: Sankey Chart

For the Sankey chart a two-step approach was used. First a simple Sankey chart was created for an understanding of the library and then a more complex chart was created. The desired results for both versions were described in detail and visualized in section 3.1.3.

4.5.1 Chart.js

The Chart.js library does not provide a Sankey chart out of the box. However, there is a JavaScript plugin available in the Node Package Manager which extends the capabilities of Chart.js to include this chart type. [60] However, this library is only written for plain JavaScript applications. In the context of this thesis, a React wrapper component was created for this plugin and published as an NPM package. [61] After installing the packages the Sankey chart can be used by importing and using the component `SankeyChart`.

The component requires a data object to be passed, which follows the format expected by `chartjs-chart-sankey` and contains both the flow data and logic for visual customization. It includes a single dataset defining the links between nodes, where each link consists of a source label (e.g., "Q1 Backlog"), a target label (e.g., "Q2 In Progress"), and a numeric flow value derived from iterating through the Pareto data. Some data transformation operations were performed to bring the data into this format. Labels on the chart are hidden by default via the `datalabels` property. To enhance visual clarity, the component uses dynamic color assignment through `colorFrom` and `colorTo`, which extract the status from each `node` and maps it to a predefined color, while `hoverColorFrom` and `hoverColorTo` provide subtle variations on hover for interactive feedback. The `colorMode` is set to "gradient" to smoothly transition between source and target colors along each link, and a `priority` mapping ensures consistent vertical positioning of nodes (always rendering "Backlog" above "In Progress").

In the sketch headings are added to the graph under each node as captions showing "Sprint 1", "Sprint 2", "Sprint 3" and "Sprint 4" which represent the time periods (often defined as 2-week periods within agile). This was not included as the library does not support adding markings within the graph.

The `chartjs-chart-sankey` plugin does not support a built-in legend because Sankey charts are structurally different from traditional Chart.js charts. Instead of representing multiple datasets with labeled series (as in bar or line charts), a Sankey chart consists of a single dataset made up of flow links connecting nodes (e.g., `{ from: 'A', to: 'B', flow: 5 }`). Since colors in a Sankey chart are dynamically assigned based on each link's source and target nodes—through custom functions like `colorFrom` and `colorTo`—Chart.js has no inherent knowledge of what these colors represent. As a result, it cannot automatically generate a meaningful legend, and a custom legend must be created manually.

The resulting first version of the Sankey chart is shown in the following figure and the implementation can be viewed online.[62]

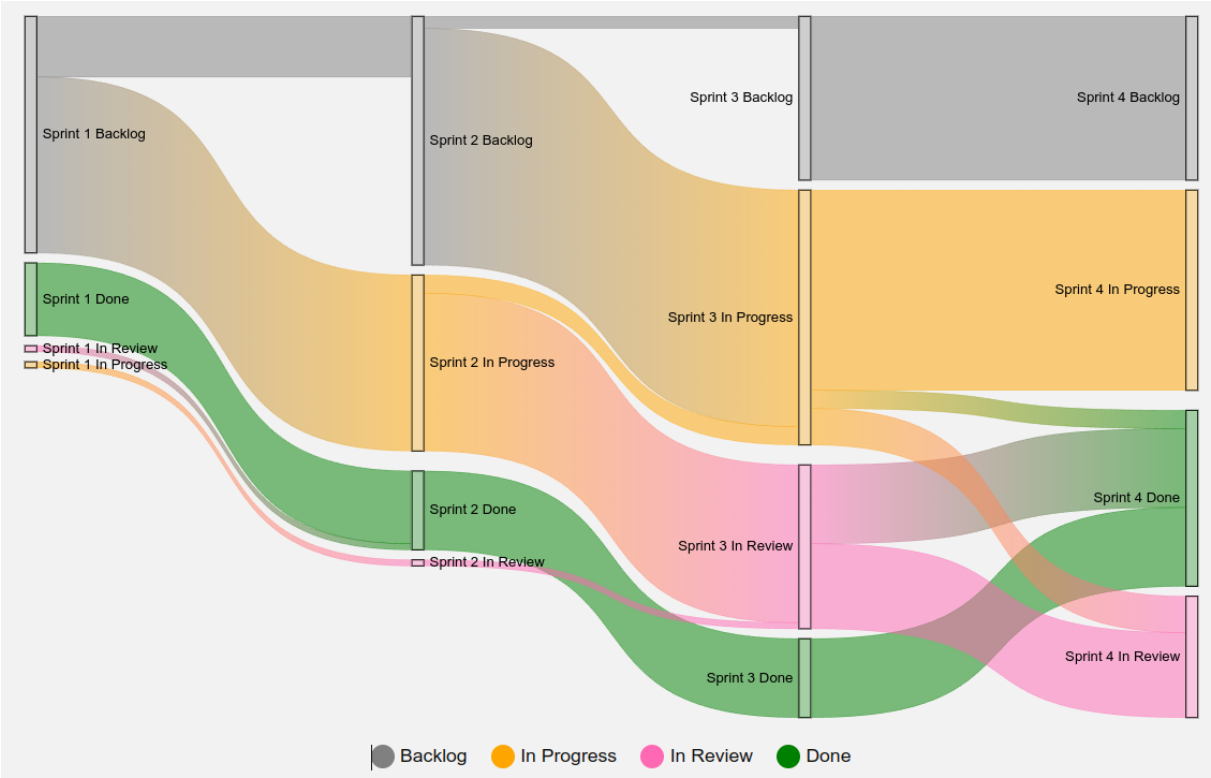

Figure 14: Sankey Chart in Chart.js

The main difference for the second version is that the links, which represent the tickets, should no longer inherit the same color as the nodes. Instead, the group to which the ticket belongs is extracted from the data. All links of groups A and B are red and C and D are gray. This was not possible with the `chartjs-sankey-chart` plugin as links always inherit colors from the source or target nodes depending on the `colorMode` setting (`gradient`, `source` or `target`).

4.5.2 ECharts

The React Wrapper `echarts-for-react` can be used for this use case as it also contains a Sankey chart. The options object passed to the `ReactECharts` component had to be adjusted. By defining the series type to be "sankey" and adding the nodes in `series.data` as well as the `links` the Sankey chart can be created.

Each node contains the values `name` as a unique identifier, `displayName` for the label displayed on the chart, `depth` to define the hierarchical level of the node, `fixed` to control whether the node's position is fixed or adjustable, and `itemStyle` to specify the visual styling, such as the color of the node.

The `links` are an array of objects containing a `source`, `target` and `value` which are derived from the Pareto dataset.

Some other settings within the series are: `layout` which is set to "none" to allow for

manual positioning of the nodes, `focusNodeAdjacency` which highlights adjacent nodes on hover, `lineStyle` which defines the color and width of the links, and `layoutIterations` an algorithm which can be used for optimizing the positioning of the nodes and links. The `label` property is used to customize the labels for both nodes and links, including font size, color, and position.

The legend is also created manually, as the library does not support a built-in legend for Sankey charts. The legend is implemented using HTML and inline styles, mapping each node status to its corresponding color. This enhances interpretability without interfering with the chart canvas.

Lastly, the graphic property in the ECharts configuration is used to add custom graphical elements to the chart. These elements can include text, shapes, images, or other visual components that are not part of the main chart series. In this case, the graphic property is used to add text annotations below the Sankey chart to label the sprints (e.g., "Sprint 1", "Sprint 2", etc.). The result of the first version can be seen in the following figure and the implementation is available online.[63]

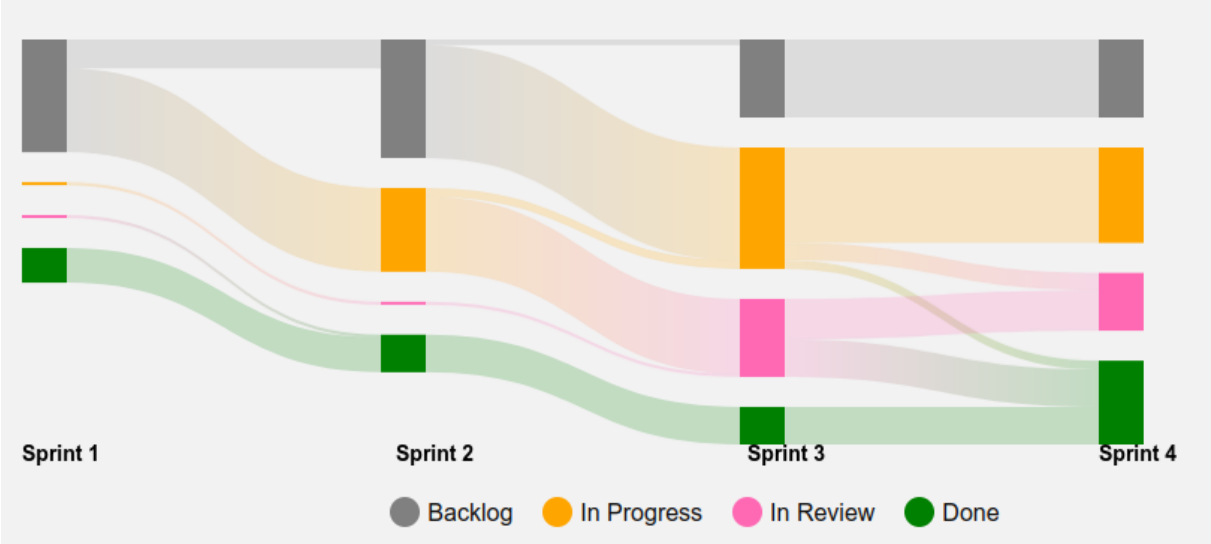

Figure 15: Sankey Chart in ECharts

The second visualization was successfully implemented with ECharts. The main change was adjusting the Sankey chart to color links based on the groups instead of the nodes, this can be achieved by adding the `lineSyle` property to the links. This property allows for individual customization of each link's appearance, including color, opacity and other visual attributes. Via some data transformation operations the groups were designated the colors red and gray and these were applied within the links.

The resulting chart is shown in the following figure and the implementation is available online.[64]

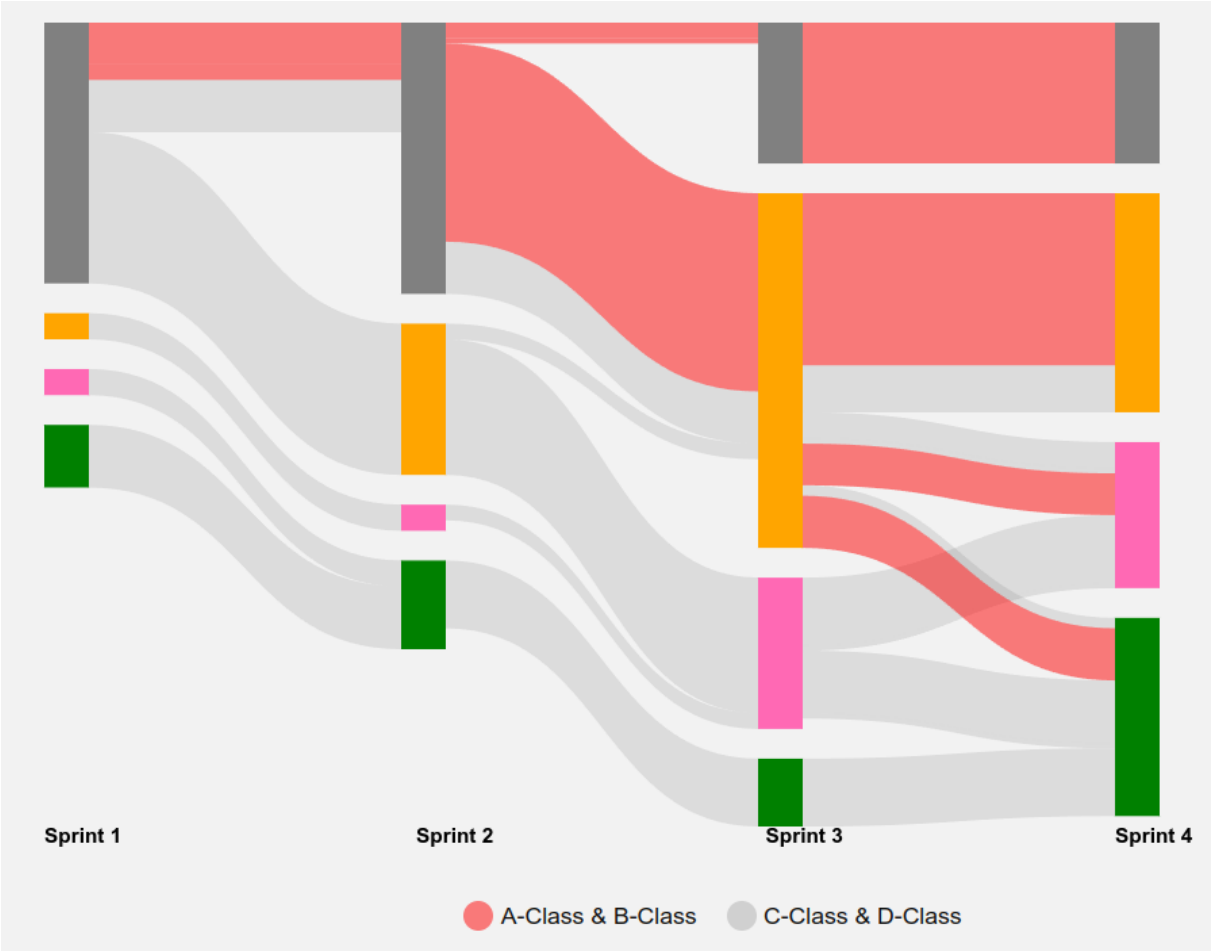

Figure 16: Sankey Chart in ECharts

4.5.3 ApexCharts

The functionality and customizability of the `apexsankey` library was also examined as part of this thesis. At the time of implementation, the library was publicly available via NPM. However, it has since been removed from public package registries. To ensure reproducibility of the results presented in this thesis, a local copy of the library has been preserved and is included in the project repository.

The library was originally available as a standalone JavaScript module. To integrate it into the React project, a `ref` was used to directly access the DOM node for manual chart initialization. The chart instance is manually created within a `useEffect` hook and initialized using a direct reference to a DOM node. `nodes`, `edges`, `options` and `plotOptions` are passed as a configuration object. Each node is defined with an `id`, a `title` and a `color`. The title is displayed at each node, however if the text size exceeds the node then it is no longer visible. The `edges` are an array of objects containing a `source`, `target`, `value` and `type`. These values are derived from the Pareto dataset. The use of `type: "curved"` ensures that edges remain distinct and traceable, even when paths overlap. Further customization is achieved via the `options` object, which defines

properties such as `edgeOpacity`, `nodeWidth`, tooltip interactivity and `edgeGradientFill`, which sets a gradient color transition from the source color to the target color for the edges.

The captions seen under the graph in the sketch are not included as similarly to Chart.js there is no inbuilt property for adding annotations/markings to the graph like the captions of the sprint number underneath the nodes.

Since the `apexsankey` library does not include a native legend component, a custom legend is implemented using HTML and inline styles. This maps each node status to its corresponding color and enhances interpretability without interfering with the chart canvas. The resulting Sankey chart is presented in the following figure. The implementation and a local backup of the library are available in the accompanying project repository for reproducibility. [65]

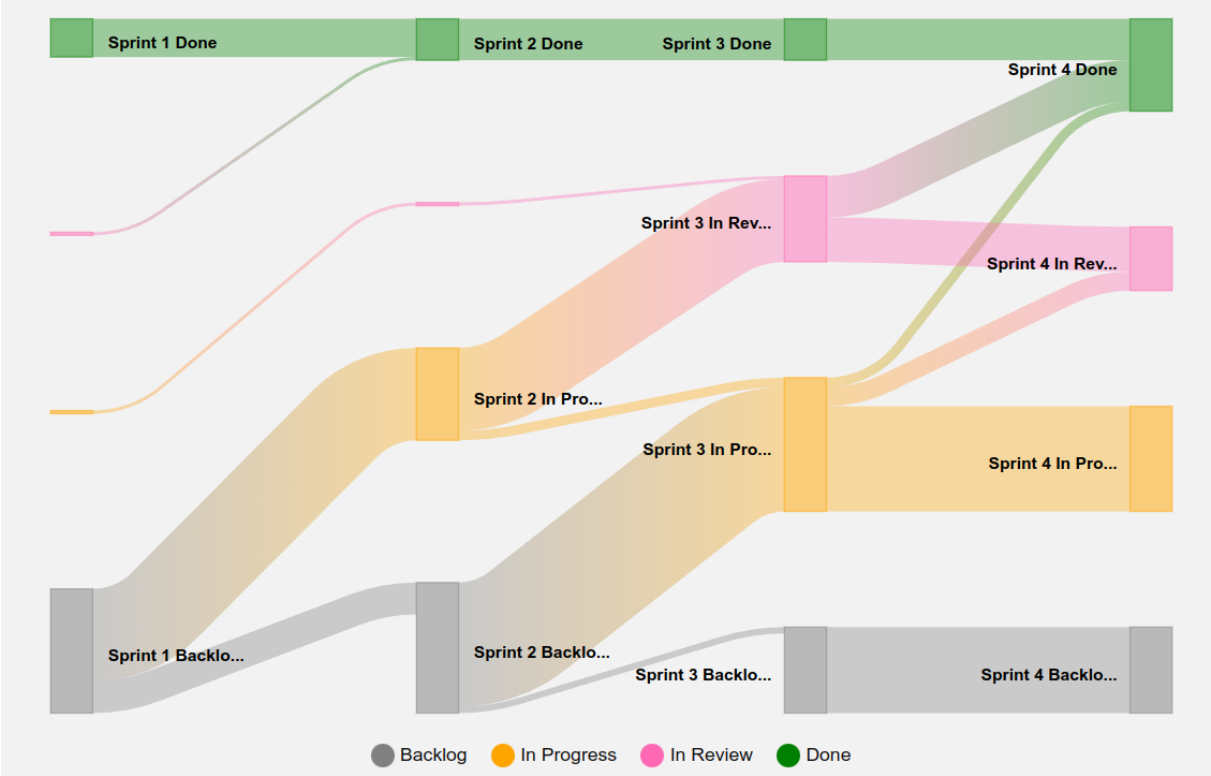

Figure 17: Sankey Chart in ApexCharts

The second version could not be implemented with the `apexsankey` library. Similarly to Chart.js, it does not support the desired functionality of coloring the `edges` independently of the `nodes`. This limitation makes it impossible to achieve the desired outcome.

4.5.4 Evaluation

ECharts emerged as the overall winner, as it was able to fulfill the requirements for both the basic and the advanced version of the Sankey chart. However, some limitations remain. For example, the library does not support a generated legend out of the box. Although

captions can be added to graph visuals, they must be manually placed. This often leads to spacing issues, as these annotations are positioned independently of the nodes, without any built-in logic for alignment or spacing. Ideally, there should be a dedicated property to assign captions to different hierarchy levels within the chart structure.

Chart.js and ApexCharts were able to implement the simpler version of the chart. However, both libraries were restricted by their data structure: individual coloring of links was not possible, as the link colors were automatically derived from the associated node colors. Additionally, neither library includes built-in support for legends. Chart.js exhibited a specific flaw: the hover tooltips did not display the correct link colors, reducing the clarity and consistency of the visual feedback. In the case of ApexCharts, another issue arose: when the label text exceeds the size of the Sankey nodes, the labels are not displayed at all. Compounding the problem, the library has since become unavailable. This introduced a new and unexpected challenge related to dependency management. To continue using the library, dependency vendoring was applied—meaning the third-party code was embedded directly into the local project and version-controlled, rather than being installed via a package manager. While this approach ensures long-term availability, it also introduces risks such as maintaining outdated code, potential security vulnerabilities, and added complexity in the codebase.[66]

5 Discussion of Results

This next section will be used to evaluate the results of the implementation. The evaluation is based on the implementation of the use cases and the experience gained during the implementation process. The results for each of the criteria will be discussed in detail in the subsequent sections and afterwards summarized within the subsection "Result Summary" at the end of this chapter. The evaluation matrix presented at the end of this chapter is based on the criteria defined in section 3.1.4

5.1 Ease of Use based on Implementation

The evaluation of the three charting libraries across three use cases—line chart, Pareto chart, and Sankey diagram—resulted in the following overall scores: ECharts performed best with 3 points, Chart.js received 2 points, and ApexCharts 1 point. These scores reflect differences in implementation effort and visual accuracy.

For the line chart use case, Chart.js provided the most efficient and developer-friendly implementation. Minimal adjustments were required to achieve the desired result, and the built-in segment property enabled straightforward differentiation of line colors. Marker customization was also easily achievable. In contrast, ECharts offered extensive customization options but demanded more complex manual configuration, such as explicitly enabling axis lines, adjusting label margins and using the color gradient property to create the multicolored line. ApexCharts, although functional, also required complex configurations for the multicolored line and marker customization, which negatively affected tooltip animations and introduced usability issues.

In the Pareto chart use case, Chart.js again achieved the most straightforward implementation. Its categorical x-axis allowed for the placement of annotations between bars without additional configuration. ECharts required switching to a numeric x-axis and manual formatting of labels, increasing implementation complexity slightly. ApexCharts demonstrated functional limitations, with hover tooltips displaying incorrect colors and incomplete annotation behavior, which could not be fully resolved.

In the Sankey diagram use case, ECharts clearly outperformed the other libraries by supporting both basic and advanced versions of the chart. This was also the only library, where the Sankey chart was in-built within the official library package, Chart.js provided this chart via a compatible plugin and ApexCharts also required the installation of a second package which is mentioned within the official website with demonstrations. [67] Although it lacked built-in legend support and required manual annotation placement, it was the only library capable of fulfilling all functional requirements. Chart.js and ApexCharts were limited by their data models, preventing independent link coloring and resulting in missing or inaccurate tooltip information. Furthermore, ApexCharts presented an additional maintenance challenge, as the library became unavailable and had to be vendored locally.

Overall, ECharts demonstrated the broadest functionality and adaptability across all evaluated scenarios, justifying its highest score. Chart.js delivered superior ease of use and stability for the line chart and Pareto chart but the Sankey chart proved to be very limited. ApexCharts was the least favorable option due to higher configuration effort, unresolved functional issues, and maintenance risks.

5.2 Responsiveness

A reusable `ChartWrapper` component was implemented to dynamically adjust chart dimensions based on user-selected sizes ("small" or "large"). The component uses React's `useState` to track the selected size and a helper function to map the size to specific width and height values. These dimensions are passed as props to the child chart components, ensuring consistent responsiveness across different chart libraries. This approach simplifies size management and effectively tests the responsiveness of the chart libraries.

All three libraries performed well in this regard, automatically adjusting the chart dimensions. For Chart.js the options property `maintainAspectRatio` was set to false, which allows the chart to resize without maintaining its original aspect ratio. The chart sizes of Echarts needed the least configuration and were correctly applied without any adjustments. Unlike the other libraries were the height and width were defined within the style attribute, in ApexCharts they are defined directly with a `height` and `width` property.

One last point to take into consideration is that for the Sankey chart all three did not have in-built legends, so a custom legend was created which is not part of the chart and therefore not responsive. This means the legend was implemented underneath the chart and is not within the height and width size restrictions of the chart.

Since all three performed equally well, they were all assigned the same score of 3.

5.3 Community and Support

Another very important factor to consider is the community and support surrounding the library. This criterion encompasses many aspects and is not easily defined or reduced to a universally accepted and measurable definition. However, there are several indicators that can be used to approximate this metric. To evaluate this, a Node.js script, "DevCareScore",[68] was created that collects relevant data from the library's GitHub repository using the GitHub REST API.[69]

The script focuses on three core metrics:

1. Issue Resolution Rate (IRR): This is the ratio of closed issues to the total number of issues (open + closed). A high IRR indicates that issues are being addressed and resolved promptly. For this calculation only the most recent 500 issues are fetched and taken into account in the calculation, since for a new user it is relevant how quickly they would receive help and an issue resolution today.

2. Median Issue Resolution Time (MIRT): This measures how quickly issues are resolved by calculating the median time between issue creation and closure. Faster resolution times generally signal better support and active maintenance. This calculation is also only based on the most recent 500 issues.

3. Contributor Count (CC): This counts the total number of contributors. A higher number of contributors suggests an active and larger developer base, which improves the chances of long-term project sustainability and user support.

   Note: The contributor count from the GitHub API may differ slightly from the number shown on the repository's web page because the API includes both authenticated and anonymous contributors, and contributors using multiple email addresses may be counted more than once.[70]

The script uses the GitHub API to fetch the data, computes the metrics, and outputs them to the console. The aim is to answer the question: if a user encounters a problem today, how quickly can they expect a response, and how likely is it that their issue will be resolved? The Contributors Count gives an indication of the size of the community. A bigger community means more people working with the library, which in turn means more people can help. This provides a quantifiable, repeatable method to assess the health and responsiveness of a library's community support—factors that are often difficult to capture through documentation or interface design alone.

Next to these quantitative metrics, the documentation needs to be considered as it is crucial for the learning curve and the overall usability of the library. Better, qualitative documentation means not having to rely on others for help or spending too much time learning through trial and error.

After working with the documentation of all three libraries, it can be said that they are all well documented and thorough, with getting started-guides or chart demos, API references, installation guides and configuration options. Since they performed equally well in this regard, the evaluation is based fully on the DevCareScore result.

Each metric of the DevCareScore has the same weight in the analysis. The output of the script showed the following results for the three charting libraries:

Table 3: Community and Support Metrics

Library	Issue Resolution Rate	Median Issue Resolution Time	Contributor Count	Result
Chart.js	43% - 2	117 days - 1	517 - 3	6
ECharts	34% - 1	51 days - 2	296 - 2	5
ApexCharts	77% - 3	4 days - 3	231 - 1	7

* The analysis was conducted on the 10.05.2025, values may change over time
** 1 = lowest-score, 2 = medium-score, 3 = highest-score; the points are summed up to receive the final score

Each library has an active community, however ApexCharts has the highest response rates and the highest resolution rate. This means a user reporting a bug, a feature request or another type of issue can expect the fastest response and the highest probability of a resolution.

5.4 Integration Flexibility

For the integration flexibility, only the integration into a React environment was evaluated. Integration with other popular frameworks like Angular and Vue.js are mentioned, though no practical experience was gained with these, and therefore they do not factor into the evaluation.

Chart.js has dedicated wrapper components available for React, Vue, Svelte, and Angular, making native integration possible. It also has built-in TypeScript typings, which are used in the implementation in this thesis. [71] However, the Sankey chart was installed via a separate plugin package which was only available for plain JavaScript. [60] In the context of this thesis, a React wrapper was written and used in the implementation. [61] This wrapper mounts the plain JavaScript Sankey plugin into the React component tree using hooks, ensuring compatibility with React's rendering cycle. It translates React props into the plugin's expected configuration, handles cleanup on unmount, and enables reactivity by updating the chart when props change. This approach allows the Sankey chart to be used like any other React component, improving maintainability, modularity, and integration with the React ecosystem. Native integration in other frameworks is however currently not available for the Sankey chart.

ECharts also has a React wrapper available.[72] There are also wrappers available for integrating ECharts in Angular [73], Svelte [74] and Vue [75]. TypeScript typings are also available, and all three chart types examined in this thesis are included in the package without the need for additional plugins or installations.

ApexCharts is available natively for React, Angular and Vue. [76] TypeScript is also supported. Similarly to Chart.js, the Sankey chart was not an inbuilt feature in the library

and an extra installation was necessary, however the Sankey chart was only available for plain JavaScript and has since been made unavailable for installation. Future plans for this chart type have been questioned but are currently unknown. [77]

Based on these findings, ECharts received the highest score of 3, as it is the only library that provides all three chart types natively in React without needing to install any further plugins. Chart.js received a score of 2, as it offers most chart types in React within the main library package however the Sankey chart is not natively supported and requires an additional installation. ApexCharts received a score of 1, as it has two of the three chart types available natively in React, but it does not provide the Sankey chart anymore.

5.5 Bundle Size

Reducing the bundle size can improve the performance and load times of web applications.[78] This is an important criterion which should also be taken into consideration. Evaluating the size of the libraries proved to be a challenge. Through initial research, a few tools were found that analyze bundle sizes. However, receiving a compact precise overview of the sizes of the library proved to be very hard because there are many factors that can influence the size, for instance most libraries are dependent on other libraries, which means transitive dependencies are also installed, different build and bundling tools have varying impacts on bundling sizes and then tree-shaking, which is mentioned in section 4.2, can reduce the size of imports but is not always included in the size calculations of these tools.

5.5.1 Import Cost

Import Cost is a Visual Studio Code extension that displays the size of the imported package in the editor. It provides a quick overview of the size of each imported module and also factors in partial imports as done by tree-shaking, helping developers make informed decisions about their code's performance and bundle size. The advantage of this tool is the instant feedback during implementation in the editor. [79] However, for evaluation purposes this tool is not suitable, since it provides feedback line for line and not on a project scale.

5.5.2 Bundlephobia

The next tool which can give insights into the size of the libraries is Bundlephobia. Bundlephobia is a web-based tool that according to the website "[...] lets you understand the performance cost of npm installing a new npm package before it becomes a part of your bundle."[80] Apart from receiving bundle size insights it also provides version comparisons, so developers can compare each release of the package, a dependency visualization and also all tree-shakeable packages are labeled. This tool is convenient to use before integrating packages into a project to get an overview of the size and dependencies.

The bundle size is specified as the minified size and the minified + gzipped size. The

minified size is the size of the JavaScript file after removing all unnecessary characters (like whitespace and comments) to reduce the file size. [81] Gzipping is a compression method that reduces the size of files for faster transmission over the web. The gzipped size is the size of the file after it has been compressed. [82] When comparing the minified sizes, Bundlephobia shows that chart.js has a size of 200kB [83], ApexCharts is larger with 555.6kB [84] and ECharts is the largest package with 1MB. [85]

5.5.3 Bundle Analyzer

To get a real-time assessment of the bundle size and the impact of tree-shaking on the application, it is advisable to use a bundle analyzer. Bundlers merge the project files into a single bundle file, which is then delivered from the server.[86] The build tool used in this thesis is vite which uses Rollup as its underlying bundler. The `rollup-plugin-visualizer` was used to generate a visual representation of the bundle.

The following table summarizes the package sizes according to the bundle analyzer:

Table 4: Package Size Comparison according to Bundle Analyser

Import Type	Chart.js	ApexCharts	ECharts
Full Package Import	482.24KB	565.88KB	2.14MB
Using Tree-Shaking	409.7KB	-	1.01MB

Installing ECharts fully without applying tree-shaking the library costs 2.14MB and when using tree-shaking it is reduced to 1.01MB. While this is a significant reduction in size, it is still the largest library of the three. ApexCharts is next with 565.88KB full bundle size with no further reduction possible since tree-shaking is not supported. Chart.js is the smallest library with a full bundle size of 482.24KB and through tree-shaking it is reduced to 409.7KB. While the figures are not directly comparable to the ones from Bundlephobia, they both come to the same conclusion.

5.6 Result Summary

For the evaluation, each criterion was assigned a weight. These weights were determined based on importance for this study. For example, ease of use was considered most important and assigned the highest weight, followed by responsiveness which was considered second most important. Community support has been weighted as the next most important criteria, since this can significantly impact long-term maintainability and ease of troubleshooting. Least important for this study were the size and integration flexibility. The size isn't as important since the main focus of this study is not performance optimization or load times in production environments. For large applications, where performance issues like high load times are a problem, this criterion would possibly receive a higher weight rating. The integration criterion is reviewed since seamless integration into a React environment can improve developer experience and reduce boilerplate code, but it is

not crucial, as JavaScript libraries can still be integrated using direct DOM manipulation. The individual scores, which range between 3 for the highest and 1 for the lowest, were multiplied by the corresponding weight and then added together to determine the final score for each library. The results of the matrix evaluation can be seen in the following table.

Table 5: Results of Evaluation Criteria*

Library	Ease of Use(4)	Responsiveness (3)	Community Support(2)	Integration (1)	Size (1)	**Result**
Chart.js	2	3	2	2	**3**	26/33
ECharts	**3**	3	1	**3**	1	**27/33**
ApexCharts	1	3	**3**	1	2	22/33

* The numbers within the table header indicate the weight of each criterion. To get the final result the weight and score of each criterion are multiplied and summed up.

The matrix table answer the first research question of this thesis:

Research Question 1: Which charting library best fulfills predefined requirements based on relevant use cases?

ECharts is the best charting library for the use cases defined in this thesis. It received the highest score of 27 points out of the total 33 points and is the only chart library that was able to implement all three use cases and thereby fulfill all requirements. The library is fully responsive, includes the highest range of chart types and is the most flexible in terms of customization. It has wrapper components available for the most popular frameworks including React, Angular, Svelte and Vue and is well documented.

A drawback for ECharts is, that out of the three examined libraries it had the worst performance in terms of community support, while it does have an active community and users receive support and qualitative documentation, it still performed the least well in terms of the average issue resolution rate and the median issue resolution time is also relatively high with 51 days. Another drawback is the package size, the library is significantly larger than the others even after applying tree shaking.

To help answer the next research question, a radar chart visualization was used to provide a more intuitive overview.

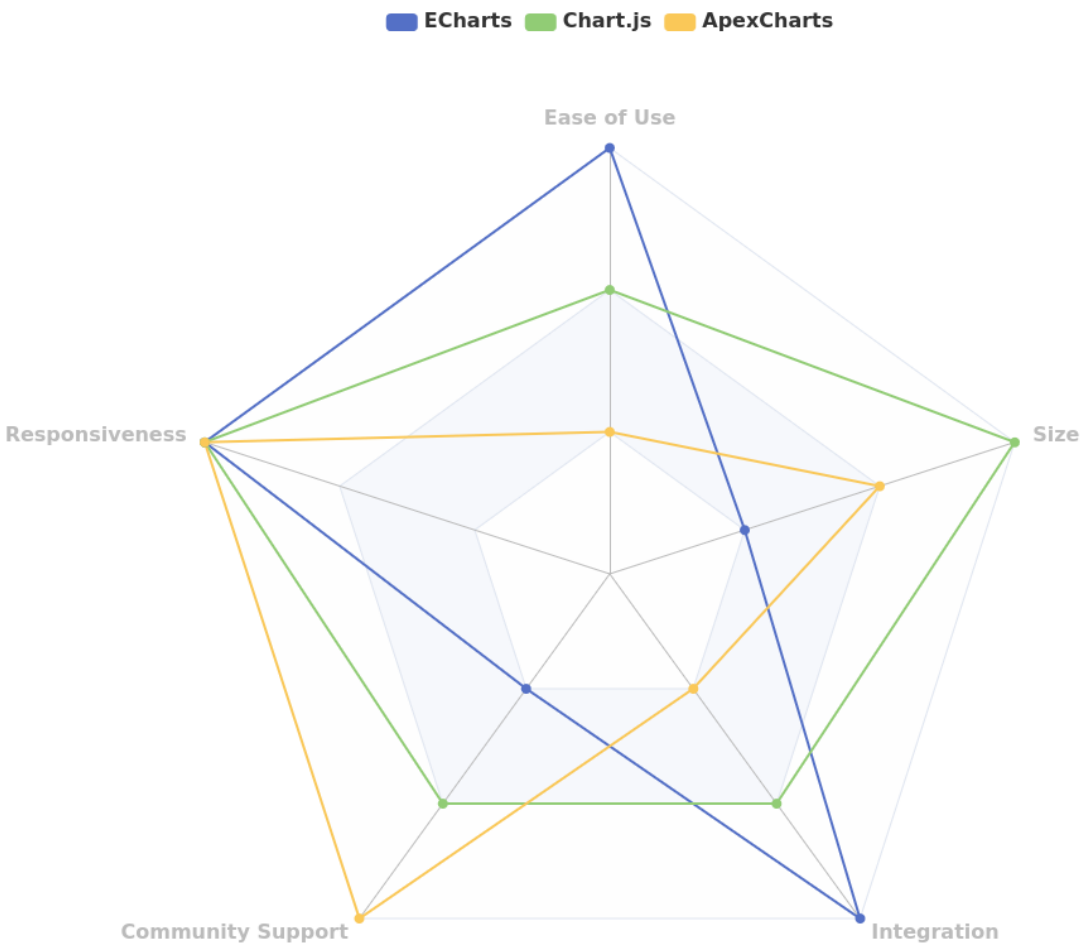

Figure 18: Radar chart of evaluation results

This chart visually represents the answer to the second research question:

Research Question 2: What are the strengths and limitations of the different charting libraries?

Each charting library demonstrates distinct strengths depending on the developer's requirements. In terms of bundle size, Chart.js is the most lightweight. For community support, ApexCharts is particularly strong, with the highest Issue Resolution Rate of 77% and a Median Issue Resolution Time of 4 days. With regard to ease of use and support for complex or customized visualizations, ECharts proves especially effective due to its extensive set of natively supported chart types and broad customization capabilities. The Sankey chart in particular demonstrated the flexibility of ECharts, as it enabled the implementation of specific visualization requirements—such as independently styled links that could not be realized with the other libraries due to limitations in their data model structures as mentioned in section 4.5.4. All libraries exhibit good responsiveness. Integration flexibility is influenced by both the chart type and the development framework; however, ECharts delivered the best performance for the React application created for this study. Across all evaluated visualization types—including bar-, line-, mixed-, and Sankey charts —the chart types are fully supported without the need for external plugins.

6 Conclusion

Faced with a vast and diverse ecosystem of charting libraries, the question motivating this thesis was deceptively simple: which library offers the best development experience? After examining multiple candidates through practical use cases and structured evaluation, this conclusion summarizes the evaluation approach used and gathers the key insights that emerged from that inquiry while taking limitations of the study into account.

6.1 Summary of the Work and Findings

First a preliminary round was undertaken to examine a broad range of charting libraries and identify those that met the core criteria of open-source availability, popularity, chart variety, documentation quality, ease of use and client-side execution. These examinations and findings are presented in chapter 2. The mentioned aspects were practically tested through basic chart implementations. Only the libraries that fulfilled all of these requirements qualified for the subsequent in-depth evaluation. Based on the insights gained from the preliminary round, Chart.js, ECharts and ApexCharts were selected for the in-depth evaluation.

Next the evaluation framework is presented in chapter 3. The three use cases of a trend line chart, a Pareto chart and a Sankey chart were defined to cover a range of chart types and complexity levels. With these use cases, the libraries are tested in practice, allowing for subsequent evaluation based on the evaluation matrix which is also introduced in the chapter.

Valuable insights into the implementation of the selected chart types and practical challenges developers may encounter are presented in chapter 4. While basic use cases were often straightforward to implement, meeting more specific or custom requirements frequently required workarounds and deep knowledge of the library internals. This included using SVG gradients for multicolored lines, injecting hidden datasets to manipulate legend content, or manipulating axis types and element positioning for precise annotation placement. These tasks revealed stark differences in each library's flexibility and robustness.

Chart.js proved to be the most capable in supporting advanced configurations through built-in features, which streamlined the implementation process. ECharts, on the other hand, demonstrated the highest degree of customizability, especially in complex scenarios such as the Sankey chart, where it was the only library that fulfilled all functional requirements. ApexCharts struggled with deeper customizations and often exhibited side effects when pushed beyond standard use cases. Furthermore, the removal of its Sankey chart during the project raised concerns about long-term reliability when relying on third-party libraries. This experience opened up a broader discussion around the trade-offs between vendoring critical components to ensure long-term availability versus relying on package registries like npm, which offer ease of use but less control over long-term access and stability.

In summary, practical implementation revealed not just the functional range of each library, but also how well they support real-world development needs. ECharts stood out

for its versatility and extensibility, while Chart.js excelled in usability for supported chart types. ApexCharts, proved less suitable for complex or heavily customized visualizations.

To ensure a well-rounded assessment, all the criteria in our evaluation matrix —ease of use, responsiveness, community and support, integration flexibility, and bundle size— were considered in chapter 5. Ease of use summarizes the implementation experience described in chapter 4. In terms of responsive sizing, all libraries performed well. It was recognized that community and support is a very broad topic and that the scope had to be narrowed down to a few key aspects. A custom tool, DevCareScore, was developed to objectively measure this criterion using three core metrics: Issue Resolution Rate, Median Issue Resolution Time, and Contributor Count based on the GitHub repositories data. These metrics offer a reproducible and data-driven method to approximate the responsiveness and health of a library's community. Additionally, the documentation was considered as a qualitative metric. Integration flexibility proved to be closely tied to how well a library supports modern frameworks and emerging chart types. Libraries that offer native React components and TypeScript support out of the box reduce integration effort and improve developer experience. Various tools are available to monitor bundle sizes, for example Import Cost or Bundlephobia. By directly using a bundle-analyser within the project, it was possible to compare the values these tools provide with those of a real-time application. It was found that tools such as Bundlephobia provide good estimates which can be used as indicators, but they can deviate from the bundle values in real-time applications. Through this practical analysis, it was recognized that libraries which allow for tree-shaking could enable significant savings in this context. Still some libraries have substantially smaller bundle sizes than others and tree-shakable libraries are not always the smallest, the libraries should be developed with saving bundle-size in mind to achieve optimal results.

Each library had its individual strengths and weaknesses, and there was not one library that outperformed the others in all aspects. By reflecting on the practical implementation experience and evaluating each criterion in the matrix, the library best suited for the purposes of this study was identified. Developers with different priorities and requirements may find another library more suitable for their needs. The radar chart provides an intuitive overview of the libraries' performance for all criteria and can be used as a reference for future projects, helping teams choose the tool that aligns best with their specific technical and organizational goals.

6.2 Limitation of the Study

The landscape of charting libraries is vast and ever-evolving, with new libraries emerging and existing ones being frequently updated and improved, that is why it should be noted that the finding of this study are a snapshot of the state of the field as of today.

It is also important to note that the evaluation matrix and the use cases were designed to be as comprehensive as possible, but they do not cover every possible scenario or requirement. Different projects may have unique needs that could lead to different conclusions.

Although the evaluation aimed to be objective and reproducible, certain criteria such

as ease of use may inherently involve subjective judgment. These experiences can vary depending on the developer's background, familiarity with the ecosystem, and personal preferences.

6.3 Contribution and Achievements

This thesis contributes to the field of data visualization by providing a structured, reproducible framework for evaluating charting libraries based on real-world implementation scenarios. The use of practical use cases in combination with a multi-dimensional evaluation matrix offers a nuanced approach that goes beyond superficial feature comparisons. Additionally, the introduction of the custom DevCareScore script and its metrics presents a novel, data-driven method for assessing community health and support—a factor often overlooked in technical evaluations.

By documenting specific customization challenges and workarounds across multiple libraries, this work also serves not only as a practical guide for developers facing similar implementation needs, but also as a critical reflection on the current state of popular visualization libraries. It highlights common limitations, inconsistent behaviors, and areas where extensibility, documentation, or support could be improved. As part of this thesis, an npm package was also created—a React wrapper for the Chart.js Sankey chart plugin—which simplifies the integration of this plugin in modern frontend projects and underscores practical aspects of library extensibility. This contribution further exemplifies how gaps in existing tools can be addressed through targeted enhancements.

The findings, tools, and methodology presented here can be reused or adapted in future studies or decision-making processes in projects requiring flexible and maintainable data visualization solutions.

6.4 Future Work

The findings and tools developed in this thesis open up several avenues for future research and practical applications. One potential direction is to extend the evaluation matrix by incorporating additional criteria. For instance, performance under large data loads could be analyzed to assess how well charting libraries scale, and support for light and dark themes could be evaluated based on how easily libraries allow for dynamic color theming or adaptation to user preferences.

Another opportunity lies in expanding the range of chart types and use cases used in the evaluation. While this thesis focused on a trend line, Pareto, and Sankey chart, further research could include other complex visualizations such as heatmaps, network graphs, or financial charts to test the flexibility and limits of the libraries under different scenarios.

Moreover, since this study was centered around React-based implementations, future work could explore how these libraries perform in other frameworks such as Angular, Vue, or Svelte, thereby providing a more comprehensive perspective on framework compatibility and integration effort.

Finally, the DevCareScore tool—originally developed as a Node.js script to assess community health—will be extended with a Python implementation to increase accessibility and enable broader adoption across different environments.

References

[1] Omorinsola Bibire Seyi-Lande et al. "The role of data visualization in strategic decision making: Case studies from the tech industry". In: *Computer Science & IT Research Journal* 5 (2024), p. 1375. DOI: 10.51594/csitrj.v5i6.1223.

[2] Venkata Siva Prasad Maddala. "Data-Driven Manufacturing: Leveraging Analytics for Operational Excellence". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 11.1 (2025), p. 885. DOI: 10.32628/CSEIT25111291.

[3] Mary C. Potter et al. "Detecting meaning in RSVP at 13 ms per picture". In: *Attention, Perception, & Psychophysics* 76.2 (2014), pp. 270–279. DOI: 10.3758/s13414-013-0605-z.

[4] *data-viz-libraries GitHub Repository*. URL: https://github.com/Deaniebean/data-viz-libraries (visited on 05/15/2025).

[5] *data-viz-libraries GitHub Pages*. URL: https://deaniebean.github.io/data-viz-libraries/ (visited on 05/15/2025).

[6] *Material Design*. URL: https://m2.material.io/design/communication/data-visualization.html#principles (visited on 03/19/2025).

[7] S. Androutsellis-Theotokis. "Open Source Software: A Survey from 10,000 Feet". In: *Foundations and Trends in Technology Information and Operations Management* 4.3 (2010), p. 189. DOI: 10.1561/0200000026.

[8] Javan Jafari et al. "Dependency Practices for Vulnerability Mitigation". In: (2023), pp. 1, 2. DOI: 10.5281/zenodo.8432714..

[9] *D3 GitHub*. URL: https://github.com/d3/d3 (visited on 03/14/2025).

[10] *What is D3?* URL: https://d3js.org/what-is-d3 (visited on 03/14/2025).

[11] *Chart.js GitHub*. URL: https://github.com/chartjs/Chart.js (visited on 03/15/2025).

[12] *HTML Canvas Graphics*. URL: https://www.w3schools.com/html/html5_canvas.asp (visited on 03/15/2025).

[13] *Chart.js Documentation*. URL: https://www.chartjs.org/docs/latest/ (visited on 03/15/2025).

[14] *Echarts GitHub*. URL: https://github.com/apache/echarts (visited on 03/15/2025).

[15] *Echarts Documentation*. URL: https://echarts.apache.org/handbook/en/best-practices/canvas-vs-svg#how-to-choose-a-renderer (visited on 03/15/2025).

[16] *Recharts GitHub*. URL: https://github.com/recharts/recharts (visited on 03/15/2025).

[17] *Recharts Official Website*. URL: https://recharts.org/en-US/ (visited on 03/15/2025).

[18] *Pyodide GitHub*. URL: https://github.com/pyodide/pyodide (visited on 03/17/2025).

[19] Ankush Joshi and Haripriya Tiwari. "An Overview of Python Libraries for Data Science". In: *Journal of Engineering Technology and Applied Physics* 5.2 (2023), p. 189. DOI: 10.33093/jetap.2023.5.2.10.

[20] *Pyodide Documentation*. URL: https://pyodide.org/en/stable/project/roadmap.html (visited on 03/17/2025).

[21] *Plotly.js GitHub*. URL: https://github.com/plotly/plotly.js (visited on 03/18/2025).

[22] *Stackgl Official Website*. URL: https://stack.gl/ (visited on 03/18/2025).

[23] *WebGL MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (visited on 03/18/2025).

[24] *Plotly.js Official Website*. URL: https://plotly.com/javascript/ (visited on 03/18/2025).

[25] *ApexCharts GitHub*. URL: https://github.com/apexcharts/apexcharts.js (visited on 03/18/2025).

[26] *ApexCharts Official Website*. URL: https://apexcharts.com/docs/installation/ (visited on 03/18/2025).

[27] *Nivo GitHub*. URL: https://github.com/plouc/nivo (visited on 03/18/2025).

[28] *Nivo Official Website*. URL: https://nivo.rocks/ (visited on 03/18/2025).

[29] *Victory GitHub*. URL: https://github.com/FormidableLabs/victory/tree/main (visited on 03/18/2025).

[30] *Victory Official Website*. URL: https://commerce.nearform.com/open-source/victory/ (visited on 03/18/2025).

[31] *MUI GitHub*. URL: https://github.com/mui/mui-x (visited on 03/18/2025).

[32] *MUI Official Website*. URL: https://mui.com/material-ui/getting-started/ (visited on 03/18/2025).

[33] Ignatius Yuwono. "5. Juice: An SVG Rendering Peer for Java Swing". In: (2006), pp. 3–4.

[34] Michail Schwab et al. "Scalable Scalable Vector Graphics: Automatic Translation of Interactive SVGs to a Multithread VDOM for Fast Rendering". In: (2006), p. 1.

[35] Jessica Piikkila. *What is SAFe?* URL: https://www.atlassian.com/agile/agile-at-scale/what-is-safe (visited on 05/05/2025).

[36] *Agile Release Train*. URL: https://framework.scaledagile.com/agile-release-train (visited on 05/05/2025).

[37] *Solution Train*. URL: https://framework.scaledagile.com/solution-train (visited on 05/05/2025).

[38] *Accelerating Flow with SAFe*. URL: https://framework.scaledagile.com/accelerating-flow-with-safe (visited on 05/05/2025).

[39] *A Deep Dive into SAFe 6.0 Flow Metrics®: Maximizing Software Delivery Efficiency*. URL: https://www.planview.com/de/resources/articles/deep-dive-into-safe-6-flow-metrics/ (visited on 05/05/2025).

[40] Daniela Kudernatsch. *Workbook Hoshin Kanri: Lean-Management-Strategien erfolgreich umsetzen*. Schäffer-Poeschel Verlag, 2022, pp. 15, 102, 113.

[41] Ethan Otto et al. "Overview of Sankey flow diagrams: Focusing on symptom trajectories in older adults with advanced cancer". In: *Journal of Geriatric Oncology* 13.5 (2022), pp. 742–746. DOI: 10.1016/j.jgo.2021.12.017.

[42] *Chart.js tree-shaking.* URL: https://www.chartjs.org/docs/latest/getting-started/integration.html (visited on 04/29/2025).

[43] *Tree Shaking.* URL: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking (visited on 03/25/2025).

[44] *Creating an Option Type in TypeScript.* URL: https://apache.github.io/echarts-handbook/en/basics/import/# (visited on 03/31/2025).

[45] *Chart.js Simple Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/thesisCode/ChartjsSimple.tsx (visited on 03/26/2025).

[46] *Chart.js API: ChartData.* URL: https://www.chartjs.org/docs/latest/api/interfaces/ChartData.html (visited on 03/25/2025).

[47] *Chart.js Custom Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/customLineChart/ChartjsCustom.tsx (visited on 03/31/2025).

[48] *Echarts Simple Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/thesisCode/EchartsSimpleLineChart.tsx (visited on 04/01/2025).

[49] *Exploring Recharts: Line Chart with different colored segments for a single Line.* URL: https://gaurav5430.medium.com/exploring-recharts-line-chart-with-different-colored-segments-8e77f13ffeb2 (visited on 04/02/2025).

[50] *Echarts Custom Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/customLineChart/EchartsLineChart.tsx (visited on 04/04/2025).

[51] *ApexCharts Simple Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/simpleLine/ApexSimpleLine.tsx (visited on 04/04/2025).

[52] *Discrete markers broken.* URL: https://github.com/apexcharts/apexcharts.js/issues/286 (visited on 04/05/2025).

[53] *ApexCharts Custom Line Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/customLineChart/ApexChartsCustom2.tsx (visited on 04/05/2025).

[54] *Chartjs Pareto Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/paretoChart/ChartjsPareto.tsx (visited on 04/09/2025).

[55] *Echarts MarkLine API.* URL: https://echarts.apache.org/en/option.html#series-line.markLine (visited on 04/10/2025).

[56] *Echarts Pareto Chart Code.* URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/paretoChart/EchartsPareto.tsx (visited on 04/10/2025).

[57] *ApexCharts Annotations API.* URL: https://apexcharts.com/docs/options/annotations/ (visited on 04/13/2025).

[58] *ApexCharts Bar Color Fix.* URL: https://stackoverflow.com/questions/71038274/how-to-set-different-color-every-bar-apexchart (visited on 04/14/2025).

[59] *ApexCharts Pareto Chart Code*. URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/paretoChart/ApexPareto.tsx (visited on 04/14/2025).

[60] *Chart.js Sankey Plugin*. URL: https://www.npmjs.com/package/chartjs-chart-sankey (visited on 04/15/2025).

[61] *Chart.js Sankey React Wrapper*. URL: https://www.npmjs.com/package/react-sankey-chartjs (visited on 04/15/2025).

[62] *Chart.js Sankey Version 1 GitHub*. URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/sankeyChart/ChartjsSankey.tsx (visited on 04/15/2025).

[63] *Echarts Sankey Version 1 GitHub*. URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/sankeyChart/EchartsSankey.tsx (visited on 04/23/2025).

[64] *Echarts Sankey Version 2 GitHub*. URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/sankeyChart/sankeyforPareto/EchartsSankey2.tsx (visited on 04/23/2025).

[65] *ApexCharts Sankey Version 1 GitHub*. URL: https://github.com/Deaniebean/data-viz-libraries/blob/main/src/charts/customCharts/sankeyChart/ApexSankey.tsx (visited on 04/22/2025).

[66] *Dependency Vendoring*. URL: https://www.activestate.com/blog/outsourced-dependency-vendoring/ (visited on 04/24/2025).

[67] *ApexCharts Sankey Chart Documentation*. URL: https://www.apexcharts.com/apexsankey/ (visited on 05/17/2025).

[68] *DevCareScore GitHub Repository*. URL: https://github.com/Deaniebean/DevCareScore?tab=readme-ov-file (visited on 05/10/2025).

[69] *GitHub API Documentation*. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28 (visited on 05/10/2025).

[70] *GitHub Contributor Discussion*. URL: https://github.com/orgs/community/discussions/24355 (visited on 05/10/2025).

[71] *Chart.js Integration*. URL: https://www.chartjs.org/docs/latest/ (visited on 05/05/2025).

[72] *Echarts React Wrapper*. URL: https://www.npmjs.com/package/echarts-for-react (visited on 05/05/2025).

[73] *ngx-Echarts*. URL: https://www.npmjs.com/package/ngx-echarts#getting-started (visited on 05/05/2025).

[74] *Svelte-Echarts*. URL: https://github.com/bherbruck/svelte-echarts (visited on 05/05/2025).

[75] *Vue-Echarts*. URL: https://www.npmjs.com/package/vue-echarts/v/6.0.0-alpha.5 (visited on 05/05/2025).

[76] *ApexCharts Chart Demos*. URL: https://apexcharts.com/javascript-chart-demos/ (visited on 05/05/2025).

[77] *ApexCharts Sankey Chart GitHub Issue*. URL: https://github.com/apexcharts/apexcharts.js/discussions/5017 (visited on 05/06/2025).

[78] *Reducing Your React Bundle Size: A Detailed Guide to Faster Apps*. URL: https://www.oneclickitsolution.com/centerofexcellence/reactjs/reducing-react-bundle-size-guide (visited on 05/08/2025).

[79] *Import Cost Visual Studio Code Extension*. URL: https://marketplace.visualstudio.com/items?itemName=wix.vscode-import-cost (visited on 05/06/2025).

[80] *Bundlephobia*. URL: https://bundlephobia.com/ (visited on 05/08/2025).

[81] *Minification*. URL: https://www.imperva.com/learn/performance/minification/ (visited on 05/09/2025).

[82] *Gzip Compression*. URL: https://www.imperva.com/learn/performance/gzip/ (visited on 05/09/2025).

[83] *Bundlephobia Chart.js*. URL: https://bundlephobia.com/package/chart.js@4.4.7 (visited on 05/09/2025).

[84] *Bundlephobia ApexCharts*. URL: https://bundlephobia.com/package/apexcharts@4.3.0 (visited on 05/09/2025).

[85] *Bundlephobia Echarts*. URL: https://bundlephobia.com/package/echarts@5.6.0 (visited on 05/09/2025).

[86] Jeremy Rack and Cristian Staicu. "Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications". In: *CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (2023), p. 3198. DOI: 10.1145/3576915.3623140.

Appendix

Overview of Used Tools

KI-Tools	Used For	With Aim	At What Time	Used Prompts
ChatGPT 4.0	Translating abstract	Ensure consistancy of abstract content	While writing	-
ChatGPT 4.0	Revising written texts	Improve the texts and ensure scientific style	After writing	-
ChatGPT 4.0	Brain-storming criteria	Find criteria to represent community and support	Before writing	How could I measure community support for different npm libraries?
SciSpace	finding scientific sources	Support literature review and strengthen the theoretical foundation	Before/ while writing	-
GitHub Copilot	In-editor code completion	Reduce development time and avoid redundant work, such as creating dummy datasets or generating boilerplate code for React components and functions	Before writing/for the imple-mentation	-