Scalable and Interactive Real-Time Visualization of Time Series Data

Bachelor's Thesis

Tim Koch

September 28, 2020

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 28. September 2020

Abstract

Time series data is everywhere. Everything that can be measured can be measured over time, which forms the definition of time series data. Visualization helps in perceiving this data by reducing the cognitive load. Interaction with the data can lead to an even better experience for users.

This thesis evaluates an approach of an interactive and scalable visualization of time series data. The approach consists of enhancing an existing visualization library and adding new features to meet the requirements. The library already supports real-time visualization and the postulated support for user interactions. To enable its scalability, a cache to reduce bandwidth load is integrated. Moreover, prefetching algorithms increase the performance of the solution.

After introducing the approach, this thesis also describes its integration into an existing web application. On this integration a feasibility and performance evaluation is conducted, which shows its advantages over the current version of the web application. Additionally, it proves the support for interactions like panning and zooming in the chart, as well as a real-time functionality. Moreover, the presented solution is scalable and does not show performance weaknesses in our test scenarios.

Contents

Chapter 1

Introduction

Time series data is a key component in engineering, business, and science. Providing visual representations for this type of data helps people to better interpret the inherent information. Line charts are widely used as such visualizations of data. To automate the process of converting time series data into charts, a programmatical approach can be valuable. Many frameworks and libraries have been published for this reason. Enabling the user to interact with the charts can improve the user experience and thus enhance the benefit of those charts. This interaction can be, for example, zooming through multiple detail levels and temporal resolutions.

The informativeness can be further improved by using real-time data. Therefore, visualization tools have to cope with the inherent challenges of processing real-time data. While providing this feature, such tools should still scale well with big amounts of data. This includes handling scaled data performantly as well as reducing the amount of displayed data to a human-perceivable level.

1.1 Motivation

The *Titan* project is a research project on how to transfer the DevOps approach into industrial production environments such as factories [Hasselbring et al. 2019]. DevOps is the concept of bundling development and operation of software closer [Lwakatare et al. 2015]. It especially aims at reducing release cycle times.

Within this project, the *Titan Control Center* serves as a platform for monitoring and analyzing the *Industrial DevOps* processes [Henning et al. 2019]. It provides an information dashboard [Few 2006] built as a single-page application with *Vue.js* [You 2014]. Among others, it uses time series charts to visualize power consumption data from the connected industrial environment. The charts are rendered using the *CanvasPlot*[1] visualization library.

The current implementation meets most of the requirements for informative time series charts mentioned above. Still, there are possible improvements to introduce. In this thesis we are aiming at the scalability and interactivity. The charts used in the Control Center are already interactive in a way that the user can zoom into the chart to see different detail levels. Despite that, the system fetches the data points to display only once, which happens on the initial page load. Because downloading all data points from the underlying database

[1]https://github.com/a-johanson/canvas-plot

would be impossible for large data sets, only data points for a specific interval are fetched. By default configuration, this interval includes the last 60 minutes. That is why older data can only be made visible by selecting a past date and time in a dedicated input form. This configuration could of course be modified to a bigger interval, but this approach would not be scalable.

The motivation of this thesis is to find and implement an approach to dynamically load data when needed. This includes loading data for other periods when zooming out, as well as loading a higher resolution of data for the given period when zooming in. The objective is to enable this feature without decreasing the performance of the chart tool.

1.2 Goals

The overall goal of this thesis is to evaluate an approach to enhance the interactivity of time series charts. Given that, the approach is to be implemented into the Titan Control Center.

1.2.1 G1: Enhanced Interactivity by Dynamic Data Loading

The existing solution including CanvasPlot is missing a functionality to dynamically load additional data points. The first goal of this thesis is to implement this feature. There are two main reasons for this goal. It enhances the user experience since the user will be able to see data for a time period longer than one hour. With that, it also supports data comparison over a longer period, which is an essential part of analyzing time series data.

1.2.2 G2: Real-Time Functionality

The current state of implementation already enables real-time visualization. While implementing 1.2.1, this behavior shall not be discarded because of its benefit for monitoring production environments in real-time.

1.2.3 G3: Scalability and Performance

When implementing the dynamic loading feature, scalability is important to keep the solution interactive under massive loads. Missing performance could render the achievements in interactivity useless. We have to assure that only a reasonable amount of data is (dynamically) loaded into the application. For example, it would inevitably lead to decreased performance, if we loaded data points for each second over periods of weeks or months.

1.2.4 G4: Integrate Approach into the Titan Control Center Frontend

After fulfilling the goals G1, G2, and G3, we will integrate a prototype into the Titan Control Center. Therefore, the prototype is integrated into a git fork of the Control Center Frontend, whereafter we create a merge request.

1.2.5 G5: Rewrite CanvasPlot as TypeScript Code

The code of CanvasPlot uses ECMAScript 5 (short ES5), a JavaScript version that does not support classes. There are multiple reasons to convert this code to a TypeScript class with ES6. Firstly, almost all of the frontend components within the Titan project are written in TypeScript. So converting the CanvasPlot would strongly contribute to a consistent code base of the whole project. Secondly, TypeScript provides the advantage of static type-checking. Moreover, the code can be cleaned up using classes and arrow functions from ES6 [Saboury et al. 2017].

1.3 Document Structure

Following this introductory chapter, we present the technologies and foundations used in this thesis in Chapter 2. Thereafter, suitable base approaches to build our solution upon are evaluated in Chapter 3. This includes identifying reasonable criteria the candidates have to meet, as well as introducing the approaches, and the final appraisal.

Then, we describe the approach for our solution in Chapter 4. We therefore determine requirements the final solution has to meet and then derive an approach from them. In Chapter 5 we describe how we integrate our approach into the Titan Control Center frontend. Following the implementation, we conduct a feasibility and performance evaluation in Chapter 6. Related work is presented in Chapter 7 before we end with drawing a conclusion and reviewing possible future work in Chapter 8.

Chapter 2

Foundations and Technologies

The following sections introduce foundations and technologies, which are used in the thesis. Section 2.1 describes the Industrial DevOps approach and Section 2.2 details the Titan platform. Moreover, the web framework Vue.js is introduced in Section 2.3. Then, the visualization library D3 and the plotting library CanvasPlot are introduced in Section 2.4 and Section 2.5 respectively. Finally, we present the TypeScript programming language in Section 2.6.

2.1 Industrial DevOps

Traditionally, the development of software is separated from its operation. This approach, however, complicates the communication, collaboration, and integration. DevOps is the concept of bundling development and operation of software closer [Lwakatare et al. 2015]. That helps developers and maintainers to work effectively and seamlessly.

Arising from the DevOps concept, Industrial DevOps is the same concept brought to industrial production environments [Hasselbring et al. 2019]. It proposes to transfer the methods and culture of DevOps and apply them to environments such as factories. To accomplish the goals of this concept, the development process in Industrial DevOps should be a cyclic, continuous adaptation process. Within the process, the system is monitored and analyzed during its operation. Then, new requirements are identified based on the analysis result. Finally, adaptions to meet the new requirements are implemented and monitored again. Crucial to the implementation phase is continuous integration to react flexibly on new requirements [Henning et al. 2019].

2.2 The Industrial DevOps Platform Titan

The goal of the Titan research project is to build a software platform for integrating and monitoring industrial production environments using Industrial DevOps [Hasselbring et al. 2019]. Skilled employees shall be empowered to modify the work flows using flow-based programming [Morrison 2010]. Thus, no dedicated programmer is needed to apply small changes in the system [Hasselbring et al. 2019].

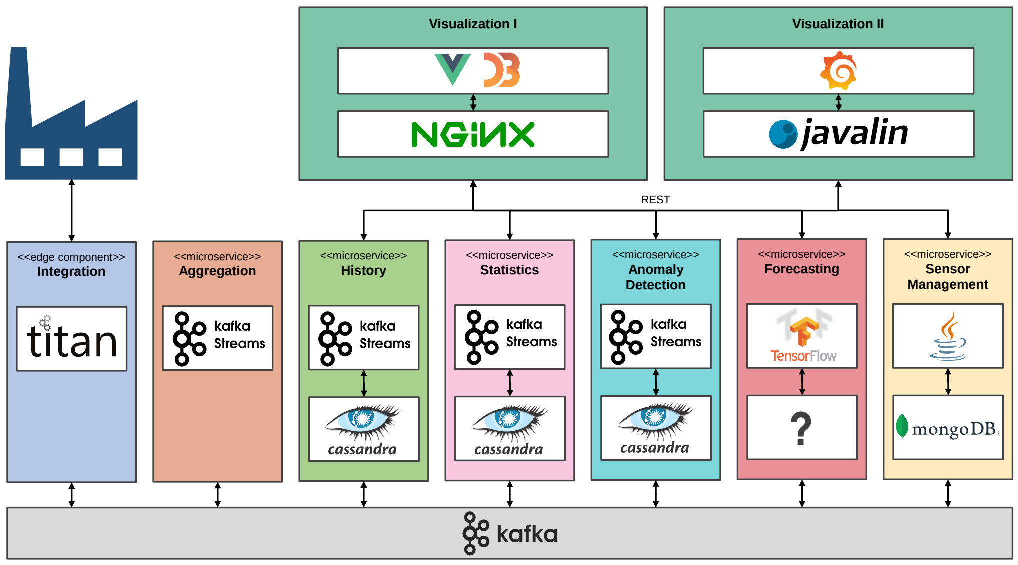

Figure 2.1. The architecture of the Titan Control Center [Henning et al. 2020]

2.2.1 Titan Control Center Architecture

Within the Titan research project, the Titan Control Center is a system for integrating, analyzing, and visualizing power consumption data from various sources within industrial production [Henning et al. 2020]. Its architecture is presented in Figure 2.1.

The system is built using the microservice pattern [Hasselbring and Steinacker 2017]. Loosely coupled services are responsible for handling different tasks that run in isolated containers and do not share any state. Power consumption data is imported using the Integration component, which converts the data into a standardized format used in the whole approach [Henning 2018].

The Aggregation service is used to group the power consumption data of multiple devices and thus provide aggregated data. Three more services conduct their own analyses on the data. While the Statistics service provides statistical information like trends in the data, the Anomaly Detection service is used for monitoring possible anomalies. Boguhn [2020] also implemented a Forecasting service.

In order to publish the recorded data within the system, the History service provides access to the database in the form of an API. Currently, four different endpoints exist. One is used for power consumption data of single devices and the second is used for group of devices. The two other endpoints are used to provide temporal aggregated data for both single and grouped devices. Those two endpoints provide minutely and hourly power consumption data. More endpoints for other temporal aggregation levels can be added.

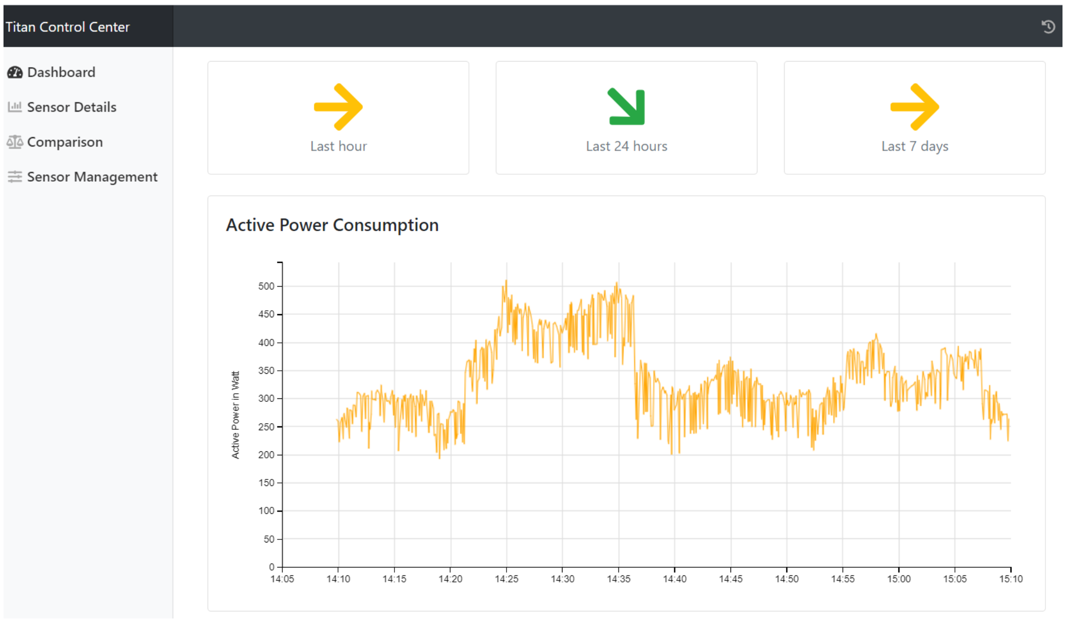

Figure 2.2. Screenshot of the Titan Control Center Frontend

2.2.2 Titan Control Center Frontend

In addition to the microservices used in the Titan Control Center and described in Section 2.2.1, the system also contains two visualization components. In this thesis, we concentrate on the first component, which we refer to as the Titan Control Center Frontend. It is a web based frontend to monitor and analyze the recorded power consumption data [Henning et al. 2019]. The single-page application built with Vue.js displays multiple charts with data for the power consumption of production environments. A screenshot of the Titan Control Center Frontend is shown in Figure 2.2.

The Current Architecture in the Titan Control Center

This paragraph introduces the architecture of the Titan Control Center Frontend. It consists of multiple pages, where three of them display time series charts. Those three pages are the landing page, a sensor details page, and a page providing a comparison chart.

The landing page and the sensor details page both delegate the chart drawing to the `SensorHistoryPlot` component, while the comparison page uses the `ComparisonPlot` component. Both the `SensorHistoryPlot` and the `ComparisonPlot` are wrapper components for the visualization library CanvasPlot, which is introduced in Section 2.5. The `SensorHistoryPlot` additionally uses the `MovingTimeSeriesPlot` as a manager to handle updates of the data displayed by CanvasPlot. An excerpt of the software architecture within the Titan Control

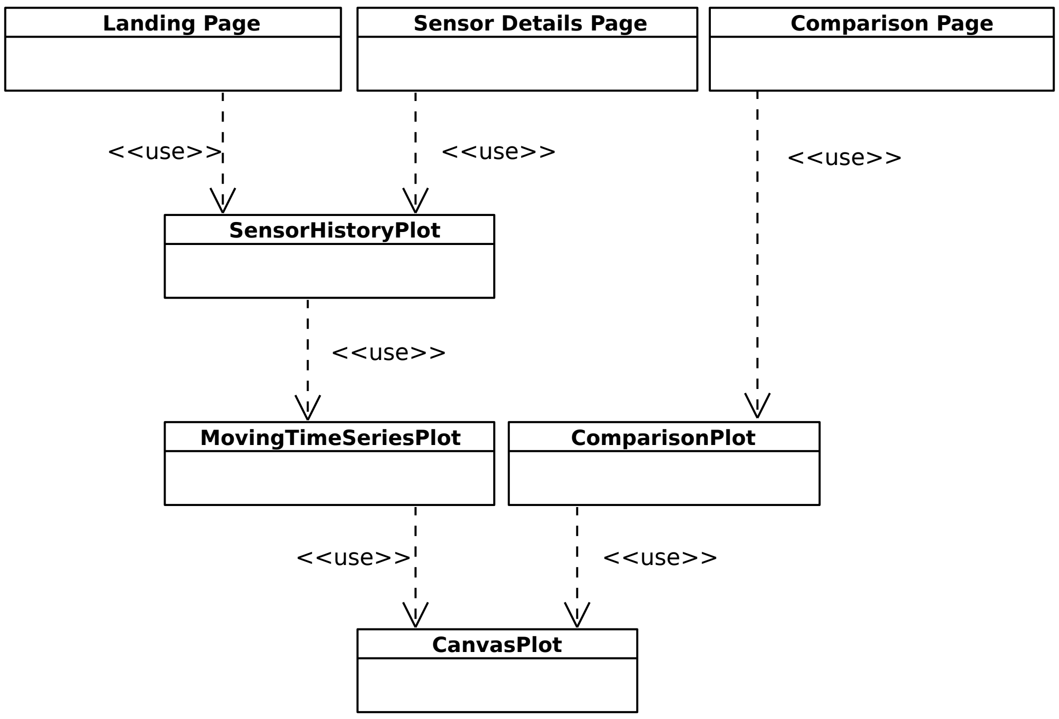

Figure 2.3. Excerpt of the Control Center Frontend architecture

Center Frontend is outlined in Figure 2.3.

The Data Flow of the ComparisonPlot

Both the `SensorHistoryPlot` and the `ComparisonPlot` described in the paragraph above handle the download of data points themselves, without any delegation to dedicated components.

The `ComparisonPlot` fetches data whenever a new device or device group is added to the comparison. It then issues a HTTP GET request against the backend. The exact endpoint depends on whether the added entity is an device or a group of devices. The query contains a filter, so that only data points from within the last hour are fetched. After the response containing the data points arrives, `ComparisonPlot` calls CanvasPlot to let it draw the new data points. This sequence is displayed in Figure 2.4.

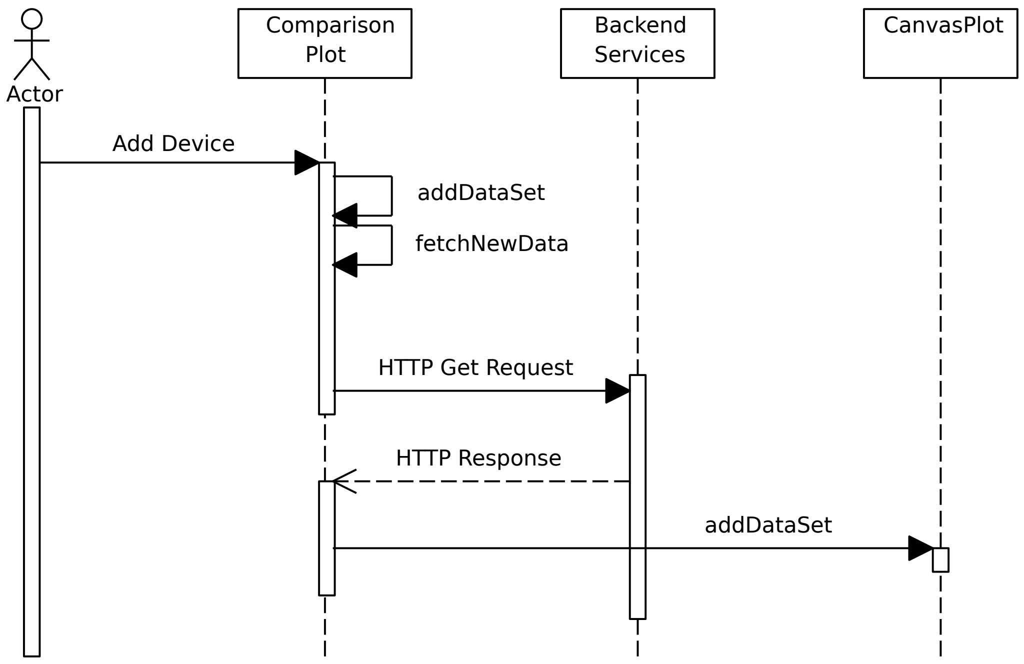

Figure 2.4. Sequence diagram for adding a device to the ComparisonPlot

The Data Flow of the SensorHistoryPlot

More logic is involved in the `SensorHistoryPlot`. Loading the page, it fetches data points similar to the `ComparisonPlot`. Which device the data is fetched for is defined externally. On the root page, aggregated data for all devices are fetched, while the device can be set specifically on the sensor details page. When the response from the backend arrives, `SensorHistoryPlot` calls its manager `MovingTimeSeriesPlot`, which then saves the data points and calls CanvasPlot to display the data.

In order to support real-time analyses of the data, `SensorHistoryPlot` uses a repeater which fetches new data points every second. These queries contain a filter that forces the backend to only respond with the data points recorded since the last fetch. The responses on those repeated queries then invoke an update of the stored data points in the `MovingTimeSeriesPlot`, which again updates CanvasPlot.

2.3 The Web Frontend Framework Vue.js

Vue.js (or just Vue) is a JavaScript framework aiming at building user interfaces [You 2014]. It does so by building single-page applications (SPA). SPAs are applications that run in the user's web browser in the form of a single web page. User interactions that lead to displaying a new page are not realized by loading a new page from a server, but instead by rendering it directly in the browser. That is why SPAs are client-side rendered applications by definition. However, these applications can still communicate with a server. As an example, this can be used to fetch more data to display. In sophisticated applications, the client can even load more chunks of JavaScript that were not included in the first response. This enables smaller and thus faster frontends.

Vue.js encourages a component-based code structure with largely independent components. That means, a Vue component includes its own logic as well as its own layout and design. This encapsulation helps making the components vastly reusable and allows a structure of composed Vue components. Components may contain other components, which leads to a hierarchical structure with one root component containing all other components [Ehrenstein 2019].

The framework also provides a sophisticated data flow model. Data exchange and thus communication is only allowed between a parent and a child component. A data flow between siblings is impossible to prevent cyclic dependencies. Moreover, children can only emit events to their parent components but are not allowed to change the parent's state.

This data flow model is also used in Vue's concept of directives. Directives allow developers to bind logic to layout elements. Thus, the state of elements can be bound to variables for example.

2.4 The Visualization Library D3

The Titan Control Center uses the data-visualization framework D3 for its charts [Bostock 2020]. Its approach is to bind data to a *Document Object Model* (DOM) and make it accessible for data-driven transformations. By making the visualization objects accessible, D3 provides flexibility and enables the control with the full tool set of web standards such as HTML, CSS and SVG. Beside that, D3 is also considered a very fast visualization tool [Bostock et al. 2011]. It also supports dynamic interactions, large data sets, and animations.

2.5 The Plotting Library CanvasPlot

CanvasPlot was built following a research on how to visualize large data sets efficiently [Johanson et al. 2016]. It is a library that extends the visualization library D3. CanvasPlot provides a simplified API to efficiently use a subset of the D3 plot types including a time series plot. Rendering underlying elements needed to display a sophisticated chart (e.g., labels or a grid) are managed by the library, which further simplifies its usage. Moreover it handles user interactions and assures a performant displaying process.

The code of CanvasPlot implements the JavaScript-typical prototypal inheritance [Eshkevari et al. 2017]. One base function *CanvasDataPlot* provides base functionality. Several

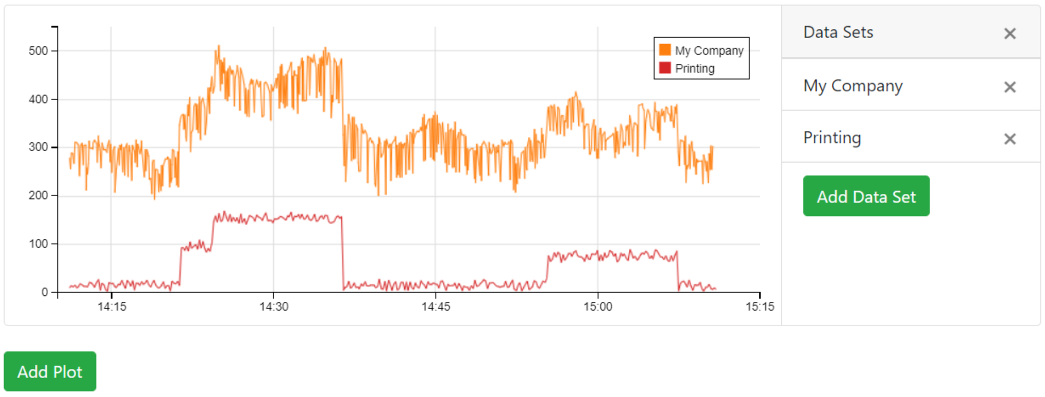

Figure 2.5. Screenshot of the ComparisonPlot within the Titan Control Center Frontend

children implement different visual representations like a time series chart or a vector diagram.

In the Titan Control Center Frontend introduced in Section 2.2.2, CanvasPlot is currently used for two different purposes. Firstly, it displays a graph representing the power consumption of either aggregated devices or single devices, depending on what the user enters in an external input field. Secondly, CanvasPlot can be used to display comparisons of either aggregated or single devices. Therefore, two or more graphs are plotted into the same chart. A screenshot of this ComparisonPlot is shown in Figure 2.5.

2.6 The TypeScript Programming Language

TypeScript is a typed superset of JavaScript [Bierman et al. 2014]. Because of this superset relation, every JavaScript program is a TypeScript program. TypeScript enriches JavaScript by adding a static type system, classes, interfaces, and a module system. However, the type system's types are optional, in order to keep the flexibility of JavaScript and to migrate from JavaScript to TypeScript in a gradual way [Feldthaus and Møller 2014]. It is to note that TypeScript's static type system does not provide strict type safety at run time. This is also to keep the system nonrestrictive.

TypeScript code can be transpiled to JavaScript code, which can run in all ecosystems that handle JavaScript. This shallow translation process converts each TypeScript expression into its representation in JavaScript.

Chapter 3

Evaluation of a Base Approach

The main goal of this thesis is to develop a solution for scalable and interactive real-time visualization of time series data. In order to fulfill this goal, we have to assess related work in this field and identify one framework to build our approach upon.

For this purpose, we elaborate evaluation criteria in order to choose the best suiting framework for our needs. Every solution is then reviewed based on these criteria.

3.1 Evaluation Criteria

We identified two groups of criteria to evaluate frameworks with. The first group consists of *framework criteria*, which allow us to assess the ability of the solutions to serve as a code framework for our goal. The second group describes the *conceptual criteria* of the given solutions. This enables us to adapt concepts from one or more previous solutions without necessarily using their code frameworks.

3.1.1 Framework Criteria

The framework criteria are used to assess frameworks based on their ability to serve as a base framework for our goals. The following criteria are summarized in Table 3.1.

License represents the license under which the framework is provided. This license should enable us to use and modify the framework free of charge.

Open Source indicates whether the source code of the framework is public. This can be useful in order to conduct minor changes, in the case of a license that allows modifying the code base.

Language defines the language of the framework. In order to fit into the code base of the Titan Control Center, a TypeScript framework is recommended.

Release Date represents the release date of the latest corresponding scientific paper. Not taking the software's release date but the release date of the paper occurred to be useful, because all of the evaluated frameworks were developed for scientific use only.

Table 3.1. Evaluation criteria

Criterion	Value Range	Requirements
Framework Criteria		
License	*License name*	non-proprietary
Open source	yes / no	yes
Language	*Language name*	TypeScript
Release date	*Date*	newer than 2015
Conceptual Criteria		
Time series capability	yes / no	yes
Interactivity	*Description of supported actions*	Zooming, Panning
Real-time capability	yes / no	yes
Scalability	*Description of scalability goal*	Support large time series data

3.1.2 Conceptual Criteria

In order to provide a more granular ranking for the base solutions, conceptional criteria are used in addition to the framework criteria.

Time Series Capability indicates whether the base approach supports the visualization of time series data.

Interactivity determines the ability of the evaluated approach to provide interactive charts. Especially the possibility to zoom through multiple resolution levels and to pan in the visual representation of the data is a key requirement for our approach. This criterion supports goal G1 (Section 1.2.1).

Real-Time Capability indicates, whether the solution is able to display real-time data. Visualizing real-time data of power consumption is a desired feature in the Titan Control Center, which is already implemented and shall be kept. This criterion supports Goal G2 (Section 1.2.2).

Scalability represents the scalability of the solution under evaluation. It is to note that this shall only decide, whether a solution aims at providing scalability. Evaluating up to which exact limit a solution scales well is not in the scope of this thesis. Scalability is here meant in terms of analyzed data, not the amount of users the system is capable to serve concurrently. This criterion supports goal G3 (Section 1.2.3).

3.2 Evaluated Approaches

In this section we present four solutions that have been used for similar use cases.

3.2.1 The System Management Visualization Tool LiveRAC

LiveRAC is a visualization system that was built to analyze large collections of system management time series data [McLachlan et al. 2008]. Its goal is to provide high information density, while allowing the users to zoom into group of charts in order to obtain a detailed insight. This zooming from condensed information into fully qualified visual data representation is called *semantic zooming*. Semantic zooming is not scaling views proportionally, but instead adding or removing properties of the visual representation. It thus balances the view on a good level of information density, between being nearly empty and being overloaded.

This technique is already used in the current implementation of the Titan Control Center. When users zoom into the provided time series charts, it begins to show circles for each data point, which disappear again on a lower zoom level. Hovering over those circles with the mouse, boxes show the x and y value of the data point. Thus, semantic zooming allows to provide more information when enough space is given, and prevents overloaded UI elements.

Unfortunately, LiveRAC does not implement semantic zooming as zooming by scrolling with a mouse or a touch pad. It allows the user to choose the shown time interval by using two date input fields. Moreover, it is scalable in a way that the application never freezes and always shows indicators when data is fetching. But it neither implements caching algorithms nor other techniques to minimize data fetching time when zooming through different detail levels.

This is why LiveRAC aims to solve the same goals as this thesis, but does not serve well as a base framework for our solution. Additionally, the solution was built using Java, which does not fit into the TypeScript code base of the Titan Control Center.

3.2.2 Interactive Visualization Tool ATLAS for Large Time Series Data

ATLAS is a visualization tool for displaying temporal data [Chan et al. 2008]. It uses several techniques to accomplish its main goal of enabling smooth interactions in massive time series data sets. The corresponding paper defines smooth interactions as fluid behaviour for operations like panning and zooming.

The researchers propose a solution consisting of three components. The database system has to store massive data sets and, more importantly, has to enable data fetching with the lowest possible latency. Every latency would impede the goal of smooth interactions. ATLAS uses kdb+, an in-memory database with high performance in time series data [Kapadiya 2018]. As the second component, a query distribution server handles the load balancing of database queries. Its job is to minimize both the query time in the database and the transfer time to the frontend.

The third component is the visual interface, which displays charts with the data fetched from the database system. Within one chart the user can zoom in, zoom out, pan left, and pan right.

Crucial to the ATLAS concept is its technique of predictive caching. It effectively hides the system latency arising from querying and fetching data from the backend. The researchers observed that interactions in the charts are likely to carry momentum, which means, for example, that users are likely to continue panning left once started. This observation is key to their predictive caching algorithm. Setting a maximum operation speed for panning and zooming, the algorithm can simply calculate when the end of the prefetched interval is reached. Based on this calculation it issues new queries to fetch data before the user reaches areas where no data is fetched yet. ATLAS also approximates the query time, which enables the system to fetch the data not earlier than it is needed, which again serves well in scalability and performance.

Because ATLAS is not a framework, we can not build our solution upon it. But instead we will take its predictive caching algorithm into consideration to benefit from its performance.

3.2.3 The Visual Exploration System ForeCache

ForeCache is a visualization system for exploring aggregate views of data [Battle et al. 2014]. The system consists of a three-tier architecture containing a visual interface, a middleware layer for caching, and a backend layer with a database management system. The corresponding paper describes how the system can be used for spatial data.

ForeCache's core concept is the usage of so-called chunks, which represent non-overlapping subsets of data. Multiple layers of chunks enable users to zoom through different resolution levels. Each layer spans over the complete data set, which means that every point in the data set is available in all possible resolution levels.

The visual interface provides a viewer for the spatial data, which allows the user to pan and zoom. Panning is possible in the four directions up, down, left, and right. Zooming in allows to display a higher resolution, while zooming out displays less details but at a greater scale. When the user conducts an operation in the viewer, the frontend issues data queries to the middleware in order to receive the data to display in the requested areas.

ForeCache's middleware receives the data queries from the frontend and again dispatches the query to the backend layer. It also serves as a cache for the system to speed up repeated queries. The backend computation layer handles the data storage of the system using a database management system. It also computes the chunks from the raw data in multiple layers.

In order to keep its interactivity, ForeCache uses prefetching to dynamically load chunks of data. It therefore works with multiple models to predict which chunks to fetch and in which order. The first model is the momentum model, which assumes that a user is likely to continue its operation in the spatial view. ATLAS uses a similar assumption [Chan et al. 2008]. Secondly, ForeCache integrates the hotspot model. This model uses tracking data of past user sessions and identifies popular chunks of data that were queried more often than others. Using this data, a hotspot data chunk to the right of the current view could be fetched earlier than a data chunk below of the current view, even if the momentum of the operation points downwards.

As a third model, ForeCache applies the n-gram model. It tries to identify probable interaction patterns by storing and analyzing past interactions as word sequences, or n-grams. ForeCache currently uses 2-length and 3-length n-grams, which could be ("down", "right") and ("down", "down", "right") respectively. Note that ForeCache is built to handle spatial data, which is why the directions represent the directions of panning in the data. The n-gram model thus tries to predict popular interaction patterns and load chunks of data accordingly.

Two more models used by ForeCache are the normal model and the histogram model. Their goal is to predict how users can move between clusters of chunks, which is not fully handled by the previous models. Both of them use statistical methods to achieve this goal. We will not discuss those models in detail because they specifically aim at spatial data and not at time series data.

Which chunks are prefetched at a specific point of time is determined by combining the results of all five prediction models. Each model ranks the possible chunks to load in a list. These lists then get merged into a final ranking, which determines the chunks to be loaded from the backend. Users may assign weights to the models, which specify their impact on the final ranking. Thus, the system provides functionality to modify the prefetch process based on personal experience.

3.2.4 The Customized D3-Wrapper CanvasPlot

CanvasPlot is introduced in Section 2.5. The scope of the following text is to evaluate it based on the criteria described above.

Users can navigate in the displayed time series charts. Zooming by mouse wheel allows them to analyze different temporal resolutions of data, however, data is not dynamically fetched based on the zoom level. It also enables semantic zooming like LiveRAC (see Section 3.2.1). In this case, CanvasPlot shows boxes with detail information on each data point, whenever there are very few data points displayed. If there are too many data points displayed to show detail boxes on each point, then those boxes disappear. Panning left and right helps to navigate within one specific temporal resolution level.

Real-time functionality is added by another wrapper, built into the Titan Control Center. The Wrapper is requesting the latest data from the server in constant intervals. This interval is currently set to 1000 milliseconds. If new data points are available they are inserted into visualization graph. The graph then moves forward if the latest data points from before the download were in the displayed interval. This ensures that the graph always shows the real-time data if the user watches it, but does not move when the user analyzes historical data. Unless otherwise stated, we refer to CanvasPlot as the core JavaScript library in conjunction with its Titan wrapper.

3.3 Evaluation of Possible Base Approaches

Table 3.2 shows an overview of the presented frameworks and solutions. Because Canvas-Plot is the only open source framework among the evaluated solutions, we will use it as a base framework for our thesis.

It is then to determine, which concepts of the other solutions shall be adapted and implemented into the Titan Control Center. Three of the four conceptual criteria are already met by CanvasPlot. It provides time series capability as well as real-time capability. The interactivity goals from Section 1.2.1 are also accomplished by CanvasPlot.

This leads to the conclusion that only those concepts have to be adopted that provide scalability enhancements to the system. Therefore, the sophisticated predictive caching models of ForeCache appear to be most-suitable for our needs. Because ForeCache uses spatial data, we will have to modify the models use time series data. This again is very similar to the ATLAS concepts, which can be adopted as well.

Table 3.2. Framework and conceptual comparison

Criterion	LiveRAC	ATLAS	ForeCache	CanvasPlot
License	*no information*	free to use	*no information*	Apache License 2.0
Open Source	no	no	no	yes
Language	Java	*no information*	JavaScript with D3.js	JavaScript with D3.js
Release	2008	2008	2014	2016
Time Series Capability	yes	yes	no	yes
Interactivity	Navigation by date input fields	Navigation by panning, zooming by mouse wheel	Navigation by panning, zooming by mouse wheel	Navigation by panning, zooming by mouse wheel
Real-Time Capability	yes	no	no	yes
Scalability	Built to analyze large time series data collections	Built to enable smooth interactions with large data sets	Built to analyze large data sets using minimal amount of data chunks	Built to efficiently display large time series data sets

Chapter 4

Visualization Approach

The following sections address the goals from Section 1.2. We start by presenting an approach for loading data dynamically based on the state of the diagram. After that, a way to maintain the real-time functionality is shown. We also describe how we manage to migrate the code base to TypeScript.

4.1 Requirements for Dynamic Data Loading

In order to provide a valid approach for the goals of this thesis, the requirements of the solution have to be identified. These requirements consist of both features existing in the current Control Center, and features yet to be implemented.

4.1.1 Features to Maintain in the Titan Control Center

Section 2.2 introduces the Titan Control Center including its frontend, while Section 3.2.4 describes the features of CanvasPlot. Our solution should maintain the current features in the Titan Control Center. In detail, this includes the device comparison chart and the chart for single or aggregated devices. Both charts should sustain their interactivity regarding the zoom and pan actions. This partly supports goal G1 (Section 1.2.1).

Additionally, the real-time capability of the `SensorHistoryPlot` should be preserved. This supports goal G2 (Section 1.2.2).

4.1.2 Features yet to be Implemented into the Titan Control Center

The requirements of the paragraph above aim at maintaining the current interactivity and real-time capability of CanvasPlot as implemented in the Titan Control Center. In order to support the remaining goals of this thesis described in Section 1.2, more features have to be implemented.

The main task is to enhance the interactivity of the current version of CanvasPlot, because it only loads data for a constant period of time. Users that analyze data from larger time periods have to set this period by hand in dedicated fields as described in Section 2.2.2. This leads to a limited efficiency in analyzing the data using the current version of the Titan Control Center.

A major requirement for our solution hence is to mitigate these limits of interactiveness within the chart. Therefore, the user has to be enabled to pan through all the data available. If this is not possible to achieve, then the user has to have at least the *impression* of panning limitlessly through the data.

CanvasPlot should also allow users to view data points aggregated over time. The current implementation already allows semantic zooming in the chart as described in Section 3.2.4. This however only enhances the charts clear look for very few data points, because it does not consolidate data points when possible. Especially in charts with thousands of data points the view becomes overloaded. Therefore, the solution has to provide a feature to choose a reasonable temporal resolution of the data displayed in CanvasPlot.

4.2 Approach for a Scalable and Interactive Visualization

After identifying the requirements in Section 4.1, this section presents the final approach of the thesis. It is divided into the architectural design, the data flow, an approach for implementing the new features, and the approach to enhance the scalability.

4.2.1 Architectural Design

Chapter 2 describes the current architecture of CanvasPlot within the Titan Control Center. This architecture is modified in order to meet the requirements made in Section 4.1. The `ComparisonPlot` is not modified and continues to serve as a visualization component for comparisons of multiple devices.

The `SensorHistoryPlot`, however, is modified substantially. It now only serves as a parent component for the CanvasPlot chart. A new class `TimeSeriesPlotManager` is introduced, which handles the data flow instead. This supports the idea of separating visualization from data handling.

The new `DataSet` class manages the available time series data in a given resolution. Multiple `DataSet` instances are stored by another new class, the `MultiResolutionData`. This class thus represents a multi-resolution data set. One instance of it is held by the `TimeSeriesPlotManager` in order to access the data.

4.2.2 Data Flow

The current data flow related to CanvasPlot within the Titan Control Center is introduced in Section 2.2.2. Between frontend and backend, the existing data flow is maintained. Henning et al. [2020] added new endpoints in the backend, in order to support temporal aggregated data. This enables the frontend to request data from the `minutely` and `hourly` endpoint.

4.2.3 Conceptual Approach to Meet the Requirements from Section 4.1

The goals of this paper are mainly accomplished by the new `TimeSeriesPlotManager`. Because it is only applied to the history plot and not to the comparison plot, it does not harm the operation of the latter. This accomplishes the requirement of maintaining the comparison plot described in Section 4.1.

In order to maintain the real-time feature of the system, the `TimeSeriesPlotManager` takes over this task from the `SensorHistoryPlot`. As the latter did before, it fetches new data every second and applies it to the chart. It then moves the chart, in the case it was set to show the latest data. This accomplishes the real-time requirement from Section 4.1 and G2 from Section 1.2.2.

These two requirements were accomplished without adding new features. So far, only refactoring took place. To meet the requirement of enabling the user to pan limitlessly through the data, dynamic data loading is necessary. Loading all data at once would not scale well, as evaluated in Section 1.2.3. That is why the `TimeSeriesPlotManager` enables a listener on the chart, which is invoked on zoom and pan actions. It then fetches the data for the displayed period of time from the backend and calls CanvasPlot to display it.

With this approach, however, CanvasPlot displays empty areas, when the user zooms out or pans left or right. This is why the time period which data is requested for, is tripled. Thus, the user can pan a whole screen length to the left or to the right without seeing empty areas. This supports interactive and efficient analyzes on the data.

The last feature requirement is to support multiple temporal resolutions. This is made possible by the new temporal aggregated endpoints implemented in the backend, as mentioned in the paragraph above. If a zoom or pan action occurs, the `TimeSeriesPlotManager` determines a reasonable resolution level and issues a query accordingly to the backend.

4.2.4 Approach to Enhance Scalability

The features required in Section 4.1 are all accomplished with the modifications described above. However, the goal of this thesis is not only to provide an interactive time series visualization, but a scalable solution as well. Requesting temporal aggregated data already reduces the amount of transferred data to an extent. This amount can be reduced even further using caching algorithms. Those algorithms enable our system to be interactive even with low bandwidths, while the load on the backend decreases as well. This supports the scalability goal of this thesis described in Section 1.2.3.

The caching algorithms of our solution are responsible for two tasks. They have to store the fetched data in a way that it is accessible for further requests. Secondly, they determine whether requested data is already cached and can be displayed without further requests against the backend. Moreover, they are able to identify requests for time periods of which subperiods are already cached. Thus, the cache can modify requests to the backend in a way that data is fetched for only those time periods, where no data is cached yet.

4.3 Approach for Migrating to TypeScript

TypeScript was introduced in Section 2.6. Because CanvasPlot has been written in plain JavaScript, it does not integrate well in the code base of the Titan Control Center. That is why we chose to migrate it to TypeScript. In order to adapt CanvasPlot to the rest of the code base, we decided to replace its prototypal inheritance with the inheritance system of TypeScript. It is to note that TypeScript's class system only hides the prototypal inheritance, because it is transpiled to exact this prototypal inheritance.

CanvasPlot was originally developed to be included in a script tag in an HTML response. That is why its code was bundled into one single file, which should be changed as well. We propose to split the code into two files based on the inheritance structure of CanvasPlot. One main file should contain the base class of CanvasPlot, while other files contain the different visual representations. Thus, the lines of code in the main file can be decreased, while the readability increases.

Chapter 5

Implementation

One major goal of this thesis is to integrate the approach described in Chapter 4 into the Titan Control Center. Therefore, we will present the implementation of our solution in a bottom-up direction. First, the migration of CanvasPlot to TypeScript is described in Section 5.1. After this, the approach for a scalable and interactive visualization from Section 4.2 is implemented in Section 5.2.

5.1 Migration of CanvasPlot to TypeScript

The approach for migrating CanvasPlot to TypeScript id described in Section 4.3. It should use the inheritance system of TypeScript and be split into two files for a better readability. Therefore, three steps of modifications are necessary, which are described in the following sections.

5.1.1 Split CanvasPlot into Two Files

The first step of migrating CanvasPlot to TypeScript is to split its code base into two separate files. This increases the readability and thus mitigates the probability of errors in the following refactoring process. In this step, we also introduce TypeScript files, because the CanvasPlot library was originally written in JavaScript. Also, the `CanvasDataPlot` has to be imported in the file containing the `CanvasTimeSeriesPlot`.

The first file contains the `CanvasDataPlot` function, while the second one contains the `CanvasTimeSeriesPlot`. Not being required for our solution, the `CanvasVectorSeriesPlot` and `CanvasDataPlotGroup` functions are deleted.

5.1.2 Introduce TypeScript's Class Inheritance for CanvasPlot

Introducing TypeScript's class inheritance is the second step in the process of migrating CanvasPlot to TypeScript. First, the `CanvasDataPlot` function is turned into a class. Therefore, a class wrapper is built, the variables of `CanvasDataPlot` are turned into class variables and the function itself is adapted to serve as a constructor for its class. The same is applied to the `CanvasTimeSeriesPlot` function.

After this, the prototype functions of `CanvasDataPlot` and `CanvasTimeSeriesPlot` are migrated into class functions of the according class. Finally, the `CanvasTimeSeriesPlot` is set to inherit from `CanvasDataPlot`.

5.1.3 Add Types to CanvasPlot

The main reason to use TypeScript is its sophisticated type system. It provides confidence in developing in larger code bases by checking type safety at compile time (see Section 2.6). Providing type safety, the TypeScript transpiler will not compile the project before we add types to our created classes.

Therefore, types are added to every class variable and to every function parameter. Additionally, a new type `DataPoint` is introduced in order to describe the data format CanvasPlot uses for its data points.

5.2 Implementation of a Scalable and Interactive Solution

We present an approach for a scalable and interactive solution in Section 4.2. After migrating to TypeScript in Section 5.1, we describe the implementation of the approach in this section. Therefore, the proposed class structure is implemented in Section 5.2.1. Then, a manager class responsible for fetching the data from the Titan Control Center History service is implemented in Section 5.2.2. In order to enable dynamic data loading, the necessary zoom handler is described in Section 5.2.3. Lastly, the implementation of the caching algorithms is detailed in Section 5.2.4.

5.2.1 Implementation of the Class and File Structure

In Section 4.2 we describe the architectural approach for our solution. The implementation of the proposed class structure is very similar. As stated in the approach, the `ComparisonPlot` remains unchanged. The `SensorHistoryPlot` however only serves as a visual container for CanvasPlot and loses its internal logic to manage and update the data in CanvasPlot. This functionality is overtaken by the `TimeSeriesPlotManager`, which is instantiated and assigned to the CanvasPlot by `SensorHistoryPlot`.

The `TimeSeriesPlotManager` uses multiple other classes. Firstly, it holds a read only reference to a `MultiResolutionData` instance, which handles the data storage for different temporal resolutions. It secondly stores a `DownloadManager` instance, which is responsible for the communication with the backend. The `DownloadManager` is described in detail in Section 5.2.2. Moreover, the `DataPoint` class is used to represent data points consisting of a date and a value. A `TimeDomain` class serves as an abstract representation of the x domain of CanvasPlot. This structure is visualized in Figure 5.1.

The `TimeSeriesPlotManager` and its used classes are bundled in a dedicated directory. An index file in this folder exports the `TimeSeriesPlotManager` class and the `DataPoint` class.

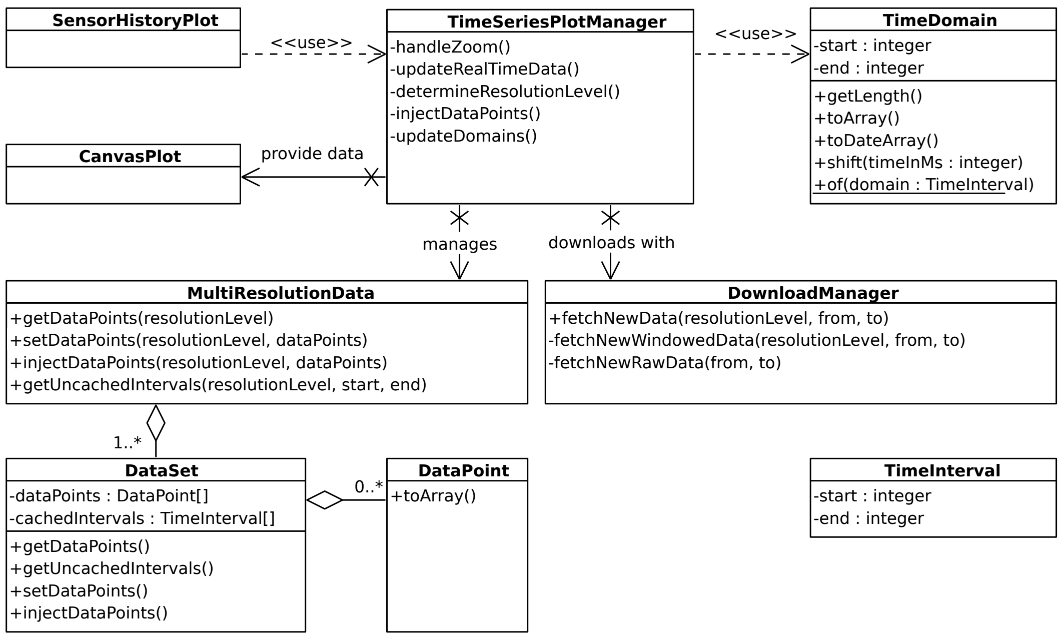

Figure 5.1. Excerpt of the implemented class structure

Only these two classes are used by other components. Index files enable developers to import components from the directory the file is in rather than from the actual component file. Thus, maintainers of this code do not have to know about the file structure within the `TimeSeriesPlotManager` directory. This supports encapsulation in our implementation.

5.2.2 Implementation of the DownloadManager

The `DownloadManager` plays a major role in our solution. It is responsible for the connection to the backend in order to fetch data points to display. As described in Section 5.2.1, the `DownloadManager` is used by the `TimeSeriesPlotManager`.

The class only provides one public function which is `fetchNewData`. As arguments it takes the temporal resolution level to fetch and the time period to fetch the data for. Based on the resolution level, the function calls either the `fetchNewRawData` function or the `fetchNewWindowedData` function. Windowed data is the term for temporal aggregated data used in the backend. Both functions take the time period to fetch data for as an argument and return promises containing the requested data points. Promises are JavaScript's way of resolving asynchronous calls.

The `fetchNewWindowedData` function additionally takes the desired resolution level as an argument. It then determines the corresponding backend endpoint for its request using a simple dictionary, which maps the numerical resolution level to URL strings.

For the `fetchNewRawData` requests the endpoint additionally differs depending on whether the power consumption of a single device or a group of devices is displayed. In the current implementation, five different endpoints are used in total by the `DownloadManager`. There are three resolution levels supported, which results in three different windowed endpoints. On these endpoints, there is no separation whether single device or a device group is displayed. Additionally, two endpoints for the raw data are supported. Those are the endpoints for separate single devices and device groups.

5.2.3 Implementation of the Zoom Handler

The zoom handler is a key part of our solution. Despite the name, it is not only called on zoom actions in the chart, but on pan actions as well. Thus, it can react on changed time periods being displayed in CanvasPlot. When the `TimeSeriesPlotManager` is instantiated and assigned to a CanvasPlot, it sets the `onZoom` callback function of CanvasPlot to its own zoom handler function.

Because a simple pan action would cause the zoom handler to be invoked once per frame, the function is debounced. Debouncing is a technique to limit function calls by not calling the original function, until a certain time passed after the last event. The currently implemented debounce time is 100 milliseconds. This prevents the zoom handler from being called more than once per user interaction.

When the zoom handler is called, it first determines the x domain of the current view in CanvasPlot. It then retrieves the start and the end of the displayed time period and calculates the span of it. By the span of the x domain, it determines the resolution to fetch. In the current implementation, three resolution levels are supported. Up to a span of 15 minutes, the zoom handler calls the `DownloadManager` to fetch raw data. If less than 11 hours are displayed, 15-minute windows are requested. Otherwise, hourly aggregated data is fetched.

After the resolution level has been determined, the `DownloadManager` is called to fetch the data. Therefore, the time period to fetch data for is tripled as a simple prefetching algorithm. When the data is received, CanvasPlot is called to display the new data points.

5.2.4 Caching the Data

In Section 5.2.2 we presented the `DownloadManager`, which is called on pan and zoom actions by the zoom handler from Section 5.2.3. It is now to determine, how the data points retrieved from the backend are added to the cache. Therefore, the `injectDataPoints` function is called when data points are received from the backend. The function takes an array of the new data points and the requested resolution level as arguments. Responsible for storing the data points is an `MultiResolutionData` class instance, which `TimeSeriesPlotManager` stores a reference to.

```
function inject (existingData: [Date, number][], toInject: [Date, number][]) {
    existingCounter, injectCounter = 0;
    resultArray = [];

    while (<resultArray not filled with all data points>) {
        existingDate = existingData[existingCounter] OR infinity;
        injectDate = toInject[injectCounter] OR infinity;

        if (existingDate < injectDate) {
            resultArray.append(existingData[existingCounter++]);
        } else {
            resultArray.append(toInject[injectCounter++]);
        }
    }
    return resultArray;
}
```

Listing 5.1. Pseudocode of the `inject` function

An instance of this data class again stores an array of `DataSet` instances. This `DataSet` represents all stored data points for a single resolution level. Hence, the multiple data sets stored by `MultiResolutionData` represent data sets for different resolution levels.

The `DataSet` provides functions to retrieve the stored data points, to set them to a new array of data points, and to inject data points into the existing array. Whenever data points are retrieved from the backend, the `MultiResolutionData` is called to inject the data points to the data set with the same resolution level. Thus, we build a multi-dimensional caching system representing the different resolution levels.

The injection of data points is done by a separate, pure function. Pure functions are those that do not produce observable side-effects [Nicolay et al. 2015]. We chose to implement a pure function here in order to ease testing, which is further described in Section 5.3.

This `inject` function takes two arrays of data points as arguments as shown in Listing 5.1. It then merges the arrays by providing two counters iterating over the arrays and choosing the older data point to add it to the new array. The resulting array is returned and stored by the data set.

In addition to storing the data points, the cached time periods are tracked as well. Therefore, the data set holds an array of time periods representing those cached periods. After new data points are injected, the `injectInterval` function is called with the stored cached periods as well as the period of the new data points. This function again returns an array of non-overlapping time periods representing the cached periods.

5.2.5 Retrieving Data Points from the Cache

Caching data is useless without retrieving it afterwards to speed up requests and decrease load on the server. The caching is detailed in Section 5.2.4. Here, we present how the data retrieval is implemented in our solution for the Titan Control Center.

Whenever the `DownloadManager` is invoked to fetch new data points, it first calls the `getUncachedIntervals` function on the data set. This function expects the time period to fetch data for as an argument. Based on the periods that have been cached before, it returns an array of intervals which represent yet uncached time periods. Afterwards, the `DownloadManager` issues GET requests only for the missing data to the backend. Thus, data is not fetched twice, which supports smooth interactions in CanvasPlot.

Because there are multiple uncached time periods possible for a single request, one `DownloadManager` call may result in multiple GET requests to the backend. All requests are promised (see Section 5.2.2) and merged into one array of data points after they resolve.

5.3 Integration of Jest as a Test Framework

Section 5.1 and Section 5.2 describe the implementation of a scalable and interactive real-time visualization for the Titan Control Center. While implementing this, we took advantage of a testing framework called Jest, in order to provide a tested and thus reliable solution.

Jest is an open-source framework built at Facebook for testing JavaScript applications [Moroz 2019]. It consists of a test runner, an assertion library, and a mocking library. Additionally, Jest is shipped with a test-coverage tool. Prior to our solution, no unit tests were used in the Titan Control Center Frontend. Tests that aim at monitoring the dependencies and static code analysis are conducted within the continuous integration process of the Titan Control Center Frontend [Latte et al. 2019].

In our implementation, Jest is not linked to any continuous integration pipeline, but it can be implemented easily in future work. This means that the test suites only run when started manually, which can be done by using a newly added script and typing `npm test`. In watch mode, Jest monitors file changes and re-runs the related tests accordingly. This mode can be invoked with `npm run test:watch`.

We used Jest in our implementation for a test-driven development regarding the two most complicated operations related to the `TimeSeriesPlotManager`. The first tested function is `injectInterval` mentioned in Section 5.2.4. The second function is called `invertedIntervalIntersections` and is used for calculating the uncached time periods in Section 5.2.5. Both functions and their tests will be presented in the following sections.

5.3.1 Testing the Injection of Non-Overlapping Intervals

We introduce the `injectIntervals` function in Section 5.2.4. It is a pure function that helps to track cached time periods. Two parameters are expected. The first is an array of time periods representing the currently cached time periods. The second parameter expects a time period representing the newly cached data. Serving as a method to track the cached time periods, its goal is to return an array of non-overlapping time periods consisting of the prior cached periods merged with the new period.

Five test scenarios have been identified. In all scenarios, two cached periods exist before merging. The details of the period to be merged are determined by the specific scenario. The new period lays before or after the existing periods, in between them, or it spans over them depending on the test case. Thus, the `injectIntervals` function is tested with the most common inputs.

5.3.2 Testing the Calculation of Uncached Time Periods

In Section 5.2.5 we describe, how data is retrieved from the cache of our solution. The `DownloadManager` calls `getUncachedIntervals` of the data set, which returns those time periods that no data is yet cached for. Calculating these uncached time periods (or intervals) means *inverting* the cached intervals in order to retrieve the uncached intervals.

The function that inverts the cached intervals is called `invertedIntervalIntersections`, because it inverts the intersections of the intervals or time periods. It was developed using a test-driven approach, which includes evaluating test scenarios first. For this function, six test scenarios have been identified. As in Section 5.3.1, two cached periods are stored and the details of the period to be requested are determined by the specific scenario. The requested period lays before or after the existing periods, in between them, or it spans over them depending on the test case. Thus, the `invertedIntervalIntersections` function is tested with the most common inputs.

Chapter 6

Evaluation

In Chapter 4 we evaluate an approach for the goals of this thesis, whereafter we describe the implementation in Chapter 5. To support the scalability goal G3, the solution and its performance must be assessed. This is the goal of the following chapter.

Section 6.1 introduces the common experimental setup for our analyses. Section 6.2 and Section 6.3 will then conduct feasibility and performance evaluations respectively.

6.1 Evaluation Environment

The tests in the following sections are all conducted in the same environment, specified in Table 6.1.

We start the Titan Control Center Frontend from the local webpack development server. The used backend services are provided by a local docker-compose cluster [Sochat 2019]. In the feasibility evaluation, sample data is used to represent power consumption data of an industrial environment. It is accessible as temporal non-aggregated data having one value every 10 seconds. Additionally, minutely and hourly aggregated data is provided.

In the performance evaluation, power consumption data of a newspaper printing company located in Kiel, Germany is used. It spans over more than 4 years and is additionally aggregated into hourly and daily data. The underlying data has a resolution of one measurement per 15 minutes which we upsampled to use minutely data.

6.2 Evaluation of Feasibility

The goal of this section is to evaluate whether the requirements described in Section 4.1 are fulfilled. We provide test scenarios in Section 6.2.1, where each consists of integration tests. Those scenarios are simulated in Section 6.2.2. Afterwards, possible threats to the validity of our results are described in Section 6.2.3.

6.2.1 Methodology

In order to test the feasibility of our approach, we conduct analyses on our implementation based on use cases. We call them scenarios in the following. A scenario consists of one or

Table 6.1. Evaluation environment

Element	Specification
CPU	Intel(R) Core(TM) i7-7700HQ
Clock Rate	2.80 GHz
Cores	4 (8 virtual)
RAM	16 GB
OS	Microsoft Windows 10 Education
OS Version	10.0.18363 Build 18363
Web Browser	Google Chrome Version 85.0.4183.83 (Official Build) (64-bit)

more subsequent user interactions, which represent typical actions in analyzing a time series chart.

The scenarios can be split in two main parts according to the requirements described in Section 4.1. Scenarios 1 to 3 aim at evaluating the features that existed before our solution was implemented and were to be maintained. Scenarios 4 to 6 shall prove that the newly implemented features fulfill their requirements. We assume that the Control Center Frontend is loaded first in every of the following scenarios and the initial data points are fetched as well.

Scenario 1: Comparing Power Consumption Data of More Than One Device

The first scenario handles a feature, which was implemented before and has to be maintained by our solution. In this scenario, we use the comparison plot to compare the power consumption of multiple devices.

To assure that graphs can be compared for multiple single devices as well as for multiple groups of devices, the comparison is done for one single device and one device group.

Scenario 2: Interact With the Plot

In the second scenario, the interactivity of the chart is tested. Maintaining support for zoom and pan actions is another requirement evaluated in Section 4.1. In this scenario, the user conducts zoom and pan actions on the chart.

We therefore navigate to the landing page of the Titan Control Center. On the upcoming chart displaying the total power consumption of the system, zoom and pan actions are conducted.

Scenario 3: Analyze Real-Time Power Consumption Data

In order to fulfill goal G2 (see Section 1.2.2), this scenario evaluates the real-time analysis capabilities of our solution. Therefore, we navigate to the landing page of the Titan Control Center. We then zoom into the chart displaying the total power consumption of the system, until the plot uses the highest-possible temporal resolution. Afterwards, we pan the chart to move the current time into view.

The plot should now move with the time passing, in order to always show the latest data.

Scenario 4: Panning Through the Data Infinitely

Scenario 4 is the first scenario to prove the feasibility of the newly implemented features as described in Section 4.1. To simulate this scenario, we conduct the same steps as in scenario 3. We then pan through the data backwards in time for at least more than 2 hours in displayed time.

This was not possible before the implementation of our solution as described in Section 2.2.2.

Scenario 5: Analyzing Data in Multiple Temporal Resolutions

In this scenario, we analyze the power consumption data in multiple temporal resolutions. We therefore navigate to the landing page in order to view the total power consumption plot of the system.

The graph should display the data in the second of three different temporal resolution levels. We then zoom into the chart in order to view the highest temporal resolution and zoom out to see the lowest resolution subsequently.

Scenario 6: Use Prefetching to Enhance User Experience

Prefetching is a technique we described in Section 5.2.3. Scenario 6 aims at testing the feasibility of this approach in our solution. Therefore, scenario 4 is simulated again, including the panning backwards in time. As long as the pan span does not exceed the width of the chart, no areas without data points should appear. This would implicate that data of a full chart length is prefetched.

6.2.2 Results and Discussion

The following section describes which observations we make while simulating the scenarios from Section 6.2.1. It also discusses whether those observations match our expectations for the given scenario.

Scenario 1: Comparing Power Consumption Data of More Than One Device

In order to simulate scenario 1, we navigate to the comparison page and click on *Add Plot*. As comparands, we choose the root device group *My Company* and a single device called *Server 1*.

Graphs for both data sets are displayed properly. Panning and zooming is supported as well. This is the behaviour we expected for scenario 1. We conclude that comparing power consumption data of more than one device is working as expected. The design did not change, hence the screenshot in Figure 2.5 is still valid for the comparison plot.

Scenario 2: Interact With the Plot

To simulate scenario 2, we navigate to the landing page in order to view the total power consumption chart. We then zoom into the chart using the mouse wheel, pan left, pan right, and zoom out again.

The chart follows all of the four interactions. While zooming, it even changes the resolution level, when a resolution limit was exceeded. This should be tested in scenario 5, but does not corrupt scenario 2. Hence we conclude that we successfully maintained the interactivity of CanvasPlot.

Scenario 3: Analyze Real-Time Power Consumption Data

Analyzing real-time data is a key requirement of this thesis presented in Section 4.1. The steps to be taken to simulate scenario 3 are described in Section 6.2.1.

The chart moves itself with the time passing, meaning that the latest data is always displayed. This is the behaviour we expected.

Scenario 4: Panning Through the Data Infinitely

In order to simulate scenario 4, we navigate to the total power consumption chart again. We then pan through the data backwards in time.

Data is fetched and displayed dynamically which results in a limitless panning backwards in time. It is only restricted by the beginning of data recorded. We conclude that infinitely panning through the data is made possible by our solution.

Scenario 5: Analyzing Data in Multiple Temporal Resolutions

As in the scenarios before, we simulate this scenario using the total power consumption chart on the landing page. Initially, the chart loads the second of three resolution levels. By zooming in, we retrieve the highest-available temporal resolution. Zooming out gives us the lowest resolution level, which currently is hourly aggregated. We hence conclude that analyzing data in multiple temporal resolutions is supported by our solution.

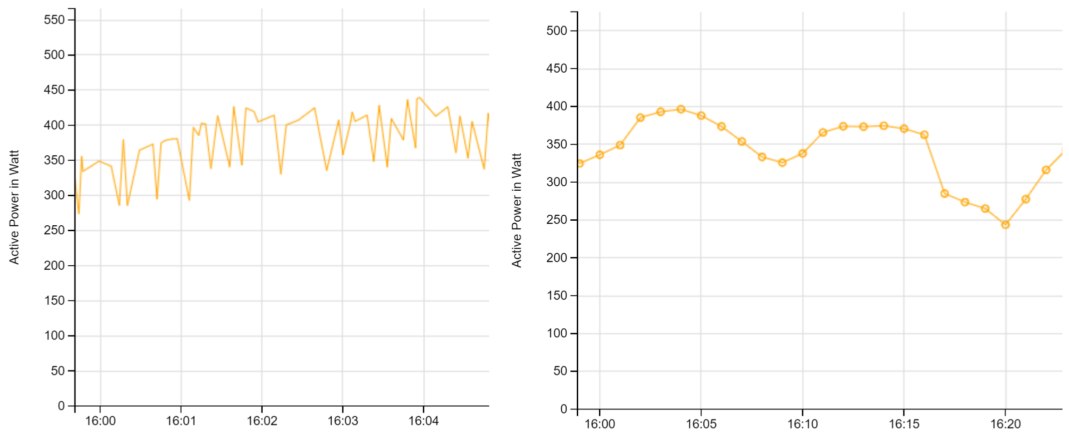

Figure 6.1. Screenshots from scenario 5: non-aggregated and minutely data

Figure 6.1 shows screenshots of scenario 5. After zooming in, the chart displays temporal non-aggregated data as shown in the left screenshot. On the right, minutely data is shown. Zooming out further would result in displaying hourly data.

Scenario 6: Use Prefetching to Enhance User Experience

Scenario 6 aims at proving the feasibility of prefetching in our solution. We therefore simulate scenario 4 again, including panning through the data.

As long as the panning span of one single pan action does not exceed the width of the chart, no empty areas are displayed. This behaviour is limited by the frequency of pan actions, the server latency, and by the download bandwidth. When multiple pan actions are conducted before the prefetched data arrives, empty areas will occur. Despite the limits, we classify this scenario as feasible.

6.2.3 Threats to Validity

For the evaluation of feasibility in this section we only used the environment described in Section 6.1. This includes the hardware environment as well as the software components and especially the Chrome browser. As a consequence, our results are only conditionally transferable to other environments since those can differ in their behaviour. Especially, the visualization was not tested in arbitrary web browsers. In general, our evaluation only comprises central scenarios and does not cover all specific edge cases.

6.3 Evaluation of Performance

After evaluating the feasibility of our solution in Section 6.2, the next section aims at evaluating the performance and scalability of our solution. In Section 6.3.1 we describe the methodology of our evaluation, followed by the presentation of the results in Section 6.3.2. Those results are then discussed in Section 6.3.3. Section 6.3.4 assesses possible threats to the validity of the evaluation.

6.3.1 Methodology

For the performance evaluation, the experimental setup is the same as in the feasibility evaluation. It is described in Section 6.1. However, the data set for this evaluation is different than the one described in other chapters of this thesis. In order to evaluate scalability and performance, we needed more data to test our solution with. Hence, power consumption data of a single device monitored from June 2015 to August 2019 with a temporal resolution of 15 minutes is used. We upsample the data to one measurement value per minute, thus our sample set contains 2,635,200 data points. Hourly and daily data is provided by temporal aggregations as well.

In our evaluation, the performance of the current implementation of the Titan Control is measured, as well as the performance of our new solution. We therefore chose to compare the load times for the chart with different amounts of data. Both versions are tested with data of 2^1, 2^2, ..., 2^{10} days. For each time span, the chart is loaded 100 times. Then, the processing time is measured and stored. This measurement contains preparing the fetch request, the response time of the backend services, the handling of the response, and the time to draw the resulting plot. Since both the frontend and the backend services run locally, the response time is not affected by delays in external networks.

We modified both of the frontend versions to automatically reload the chart and to double the requested time period after 100 measurements. They also calculate the processing time themselves by storing two timestamps. The first is set to when the fetch is prepared and the second timestamp stores the current time when the chart is fully drawn. Then, the difference is taken as the final measurement. Those measurements are then sent to a server specifically created to receive these values and to store them in CSV files for later analyses.

6.3.2 Results

The measurements of the evaluation are plotted in Figure 6.2 and Figure 6.3, displaying the results of the current implementation and the new implementation respectively. These boxplot diagrams show the processing time in milliseconds for fetching and drawing data for the given amount of days. Each box represents the measured processing times lying between the first and the third quartile. Thus, it represents 50% of the sample. Moreover, the vertical lines originating from the boxes called whiskers indicate the total range of

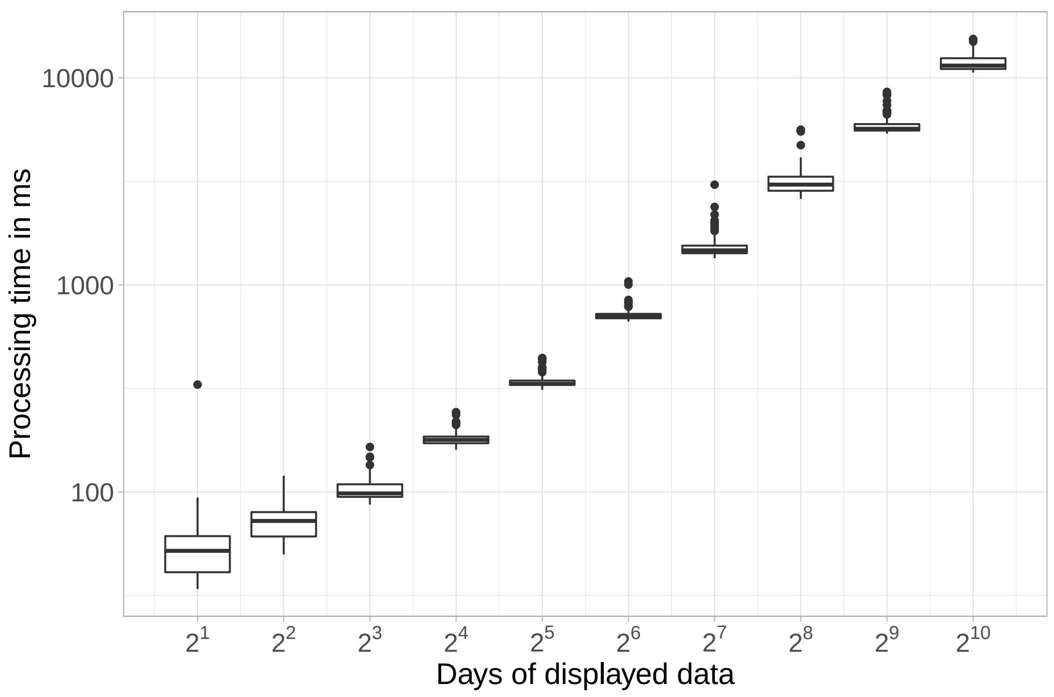

Figure 6.2. Time to fetch and draw data for given number of days - current implementation

the sample. The upper end of the lines hence marks the maximum value of the sample, while the lower end marks the minimum value. Points above the upper whisker represent statistical outliers. Values are considered outliers, if they are greater than the 3rd quartile plus 1.5 times the range between the second and third quartile.

It is to note that we use logarithmic scales in the charts, in order to enhance the visualization of exponentially growing time spans to fetch data for. While the x axis uses a base-two logarithmic scale, the y axis opts for a factor of ten.

Figure 6.2 visualizes the results for the current version. It shows that the processing times increases when the number of days to display data for grows. While it takes less than 100 milliseconds to fetch and draw data points for 4 days and less, the same process takes more than 10 seconds for 1024 days. We can presume that the growth of the processing time is linear to the days of displayed data.

The results of the new solution, however, do not show linear growth, which is shown in Figure 6.3. For 2, 4, and 8 days of displayed data, the median processing time lies around 20 milliseconds. Processing 16, 32, and 64 days of data takes around 14 milliseconds. For more days to display, the processing time then rises until it reaches a median of 32 milliseconds when displaying 1024 days.

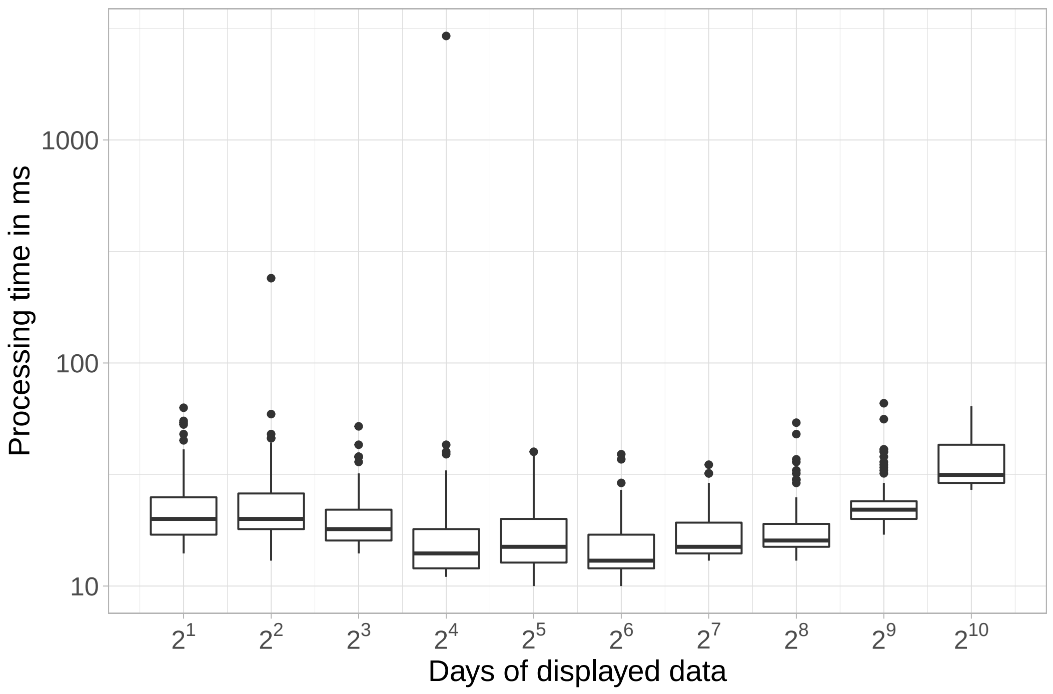

Figure 6.3. Time to fetch and draw data for given number of days - new implementation

6.3.3 Discussion

The performance results described in Section 6.3.2 for the current implementation and our new solution both meet our expectations. Because the current version does not fetch temporal aggregated data, the linear growth of processing time is reasonable. Two time periods result in the same processing time, if the time span is equal. Hence, the processing time grows according to the time period.

The results from the new solution shown in Figure 6.3 can also be explained by the underlying approach. For the performance evaluation, the default resolution level for the displayed data was changed in order to handle the hourly and daily data. Therefore, the highest available resolution is only requested if the time span to display is less than one day. If data for more than one day is fetched but for less than two weeks, than hourly data is requested. Otherwise, daily data is downloaded from the backend services. This change assures that only a reasonable amount of data is fetched and displayed.

This switching between resolutions explains why the measurements are smaller than the ones from the current implementation. Minutely data as used by the current version is never fetched by the new solution, because we always measure the processing time for more than one day. The decrease between 8 and 16 days of displayed data is triggered by the change from hourly to daily data, because data for more than two weeks is fetched. Since no resolution level greater than daily data is provided, the processing time for more days then rises as expected.

The decrease of processing time from 2^5 to 2^6, however, is surprising, because there is no possible change in resolution levels. This can be caused by measurement error, which again could be caused by multiple reasons. Those reasons are described in the following section.

6.3.4 Threats to Validity

The environment we run the performance evaluation in is not specifically designed to support a microservice architecture like the Titan Control Center. Moreover, all components of the Control Center are executed on the same physical machine. Those components hence compete for computational resources on this machine. Additionally, the evaluation is run in a desktop environment, which adds even more processes to run in parallel to our test setup. These reasons can lead to measurement errors in our evaluation.

Chapter 7

Related Work

The solution in this thesis is built upon the Titan Control Center introduced by Henning [2018]. It is responsible for monitoring the electrical power consumption within the Titan platform (see Section 2.2). Our solution also uses CanvasPlot by Johanson et al. [2016] for visualizing the power-consumption using line charts. CanvasPlot itself wraps the open-source data visualization framework D3 by Bostock [2020]. The Titan Control Center, CanvasPlot, and D3 are introduced in Chapter 2.

The information arising from monitoring industrial environments using the Control Center is a key component to support the Industrial DevOps approach proposed by Hasselbring et al. [2019]. A coherent circle consisting of monitoring and analyzing, identifying requirements, adapting the system, and automatic testing before deploying the software into production is introduced by the authors of the paper. Additionally, a concept for an organizational structure is presented. It shall support quick adjustments to the system by encouraging people from different domains to work together.

In Chapter 3 we evaluate existing approaches for scalable and interactive time series visualization in order to identify a base approach for our solution. Thus, four related approaches are presented in Section 3.2. LiveRAC, ATLAS, ForeCache, and CanvasPlot are visualization libraries that all aim at supporting large sets of time series data.

LiveRAC is introduced in Section 3.2.1. It is a tool providing visualizations for system management time series data [McLachlan et al. 2008]. The strengths of this library is its use of semantic zooming which means not to zoom proportionally but adding or removing properties of the visual representation. This technique helps balancing the information density on a well-defined level. A US patent for semantic zoom filed is abandoned but still assigned to Microsoft Technology Licensing LLC [Pittappily 2013]. In contrast to our approach, LiveRAC does not implement any caching algorithms or other techniques to minimize data exchange volume. Moreover, it does not support pan actions in its charts.

ATLAS however is not built for a specific type of time series data [Chan et al. 2008]. Its goal is to maintain interactivity of time series plots while displaying large data sets. In detail, the authors of the corresponding paper define interactivity in supporting smooth interactions like panning and zooming. In our thesis, we adopted their goal in conjunction with its definition.

While we focus on presenting a frontend meeting our requirements, ATLAS additionally sets up a high-performance server architecture. The system consists of the frontend itself, a query distribution server acting as a load balancer, and a kdb+ database specifically built for time series data. ATLAS introduces the concept of predictive caching which hides the latency of the system by prefetching data. It therefore anticipates which data might be viewed after the next user interaction and then requests it from the database. For this thesis, we adopted this concept to enhance the interactivity of our solution.

ForeCache as the third of the evaluated approaches adds the concept of using multiple prediction models in the prefetching process [Battle et al. 2014]. Those models all make assumptions about which data is viewed next and thus define which data is prefetched first. For example, one model assumes that a user action like panning will continue rather than stop immediately and thus proposes to prefetch the data that lies in the direction of the pan action. Another model works with recorded user sessions to identify so-called hotspots in the data which are more likely to be viewed than other spots. For our implementation we implemented the first model only, which proves sufficient in the performance evaluation in Section 6.3. In contrast to the other evaluated libraries and our approach, ForeCache does not display time series data but spatial data instead. Its approach to prediction models, however, can be valuable for time series data as well.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we presented an approach for scalable and interactive visualizations of time series data. We therefore first explained our motivation for this field of work and identified the goals for this thesis. Thereafter, we introduced the foundations and technologies that were used in our solution. In order to build our implementation on a reliable base approach, we assessed four visualization libraries based on framework and conceptual criteria. In this evaluation, we identified CanvasPlot as such a base library for our solution.

We then constructed several requirements which our implementation had to meet in order to support all goals described in the first chapter. Based on them, we introduced our approach for an interactive and scalable visualization of time series data. It consists of using CanvasPlot as a base solution, maintaining its features like real-time capability and support for zoom and pan actions, and adding more abilities. These include the support for temporal aggregated data, caching, and prefetching. We also presented an approach for migrating CanvasPlot to TypeScript.

After having finalized the approach, we began with the implementation by migrating to TypeScript. Then, we introduced the new class and file structure within the Titan Control Center Frontend to meet the requirements. A download manager was implemented to fetch data from the Control Center backend services. The zoom handler was introduced in order to react on zoom and pan actions conducted by the user. To meet the scalability goal of this thesis, we also integrated caching algorithms.

Following the implementation and integration into the frontend, an evaluation of our solution was conducted. We therefore evaluated its feasibility in six scenarios which all showed positive results. Then, we tested the performance of the current version of the frontend and our new approach. Therefore, we measured the time to fetch and draw data points for numerous time spans, which clearly showed an advantage of the new solution.

We provide our implementation as a package [Koch 2020] that contains the Titan Control Center Frontend including our visualization solution from Chapter 5. It also includes the versions tested in the performance evaluation, as well as the used log server. In order to provide backend services for those versions, we append a docker-compose file to start all necessary Titan Control Center services.

8.2 Future Work

During our work on this thesis, we identified multiple aspects that can be considered in future work. These aspects can be classified by their proximity to our solution and the Titan research project.

8.2.1 Future Work within our Solution

Our solution meets all the goals we define in Section 1.2. However, the evaluated approaches encourage to go further. Especially ForeCache's use of multiple prediction models to determine which data is prefetched could be interesting for future enhancements of our approach [Battle et al. 2014]. As described in Section 3.2.3, it predicts the data to be viewed next by consulting five different prediction models.

Most of them aim at spatial data instead of time series data, but the so-called hotspot model could yield interesting results in our solution too. It works by recording user sessions and storing the most often displayed data. Then it assumes that those hotspots will be viewed more often in future user sessions and will prioritize them in the prefetching process. Future work can integrate this feature in order to enhance the prefetching proposed in this thesis.

8.2.2 Future Work within the Titan Control Center

The evaluation in Chapter 6 shows that the scalability of our approach depends on the temporal resolution levels of the time series data provided by the backend services. This leads to the need of more such endpoints when the data set grows in order to maintain the level of performance observed in this thesis.

In the current and in the new implementation, CanvasPlot repeatedly requests the latest data from the backend services in constant intervals. Thus, the visualization latency includes the time until the next logical tick of the timer when the next data fetch is conducted. This latency hence can be diminished by introducing an implementation of the WebSocket protocol [I. Fette and A. Melnikov 2011]. Using this protocol allows server-sent messages containing the latest data which could be sent immediately after the values are received and processed by the backend services. After implementing this technique we can expect a vastly decreased latency [Pimentel and Nickerson 2012].

8.2.3 Future Work within the Field of Scalable Visualizations

Outside of the Titan context, a scientific evaluation could be conducted to test the scalability limits of a solution combining the approaches of ATLAS and ForeCache. From ATLAS one could adopt its query distribution server and the high-performance database kdb+ which is specifically designed to store large time series data sets. Adding ForeCache's prediction models could result in a system with astonishing performance benchmarks.

Bibliography

[Battle et al. 2014] L. Battle, R. Chang, and M. Stonebraker. Dynamic generation and prefetching of data chunks for exploratory visualization. *IEEE InfoVis Posters Track* (2014). (Cited on pages 16, 44, 46)

[Bierman et al. 2014] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In: *ECOOP 2014 – Object-Oriented Programming*. 2014. (Cited on page 11)

[Boguhn 2020] L. Boguhn. Forecasting power consumption of manufacturing industries using neural networks. Bachelor's Thesis. Kiel University, Department of Computer Science, 2020. (Cited on page 6)

[Bostock et al. 2011] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics* (2011). (Cited on page 10)

[Bostock 2020] M. Bostock. *D3.js - data-driven documents*. Library Catalog: d3js.org. URL: `https://d3js.org/` (visited on 05/02/2020). (Cited on pages 10, 43)

[Chan et al. 2008] S.-M. Chan, L. Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In: *2008 IEEE Symposium on Visual Analytics Science and Technology*. 2008. (Cited on pages 15, 17, 43)

[Ehrenstein 2019] S. Ehrenstein. Distributed sensor management for an industrial DevOps monitoring platform. Bachelor's Thesis. Kiel University, Department of Computer Science, 2019. (Cited on page 10)

[Eshkevari et al. 2017] L. Eshkevari, D. Mazinanian, S. Rostami, and N. Tsantalis. JSDeodorant: class-awareness for JavaScript programs. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017. (Cited on page 10)

[Feldthaus and Møller 2014] A. Feldthaus and A. Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*. 2014. (Cited on page 11)

[Few 2006] S. Few. *Information dashboard design: the effective visual communication of data*. 2006. (Cited on page 1)

[Hasselbring et al. 2019] W. Hasselbring, S. Henning, B. Latte, A. Mobius, T. Richter, S. Schalk, and M. Wojcieszak. Industrial DevOps. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019. (Cited on pages 1, 5, 43)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017. (Cited on page 6)

[Henning 2018] S. Henning. Prototype of a scalable monitoring infrastructure for industrial DevOps. Master's Thesis. Kiel University, Department of Computer Science, 2018. (Cited on pages 6, 43)

[Henning et al. 2020] S. Henning, W. Hasselbring, H. Burmester, A. Möbius, and M. Wojcieszak. Goals and measures for analyzing power consumption data in manufacturing enterprises. *arXiv:2009.10369 [cs]* (2020). (Cited on pages 6, 22)

[Henning et al. 2019] S. Henning, W. Hasselbring, and A. Mobius. A scalable architecture for power consumption monitoring in industrial production environments. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. 2019. (Cited on pages 1, 5, 7)

[I. Fette and A. Melnikov 2011] I. Fette and A. Melnikov. *The WebSocket protocol*. RFC 6455. RFC Editor, 2011. (Cited on page 46)

[Johanson et al. 2016] A. Johanson, S. Flögel, C. Dullo, and W. Hasselbring. OceanTEA: exploring ocean-derived climate data using microservices. In: *Proceedings of the Sixth International Workshop on Climate Informatics (CI 2016)*. 2016. (Cited on pages 10, 43)

[Kapadiya 2018] A. A. Kapadiya. Visualisation of multi-service system net-work with d3.js & kdb+/q using websocket. *Global Journal of Computer Science and Technology* (2018). (Cited on page 16)

[Koch 2020] T. Koch. *Thesis Artifacts for: Scalable and Interactive Real-Time Visualization of Time Series Data*. 2020. URL: `https://doi.org/10.5281/zenodo.4041682`. (Cited on page 45)

[Latte et al. 2019] B. Latte, S. Henning, and M. Wojcieszak. Clean code: on the use of practices and tools to produce maintainable code for long-living software. *Living Systems* (2019). (Cited on page 30)

[Lwakatare et al. 2015] L. E. Lwakatare, P. Kuvaja, and M. Oivo. Dimensions of DevOps. In: *Agile Processes in Software Engineering and Extreme Programming*. 2015. (Cited on pages 1, 5)

[McLachlan et al. 2008] P. McLachlan, T. Munzner, E. Koutsofios, and S. North. LiveRAC: interactive visual exploration of system management time-series data. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2008. (Cited on pages 15, 43)

[Moroz 2019] B. Moroz. Test automation of a react-redux application with jest and enzyme. Bachelor's thesis. South-Eastern Finland University of Applied Sciences, 2019. (Cited on page 30)

[Morrison 2010] J. P. Morrison. *Flow-based programming, 2nd edition: a new approach to application development*. 2010. (Cited on page 5)

[Nicolay et al. 2015] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter. Detecting function purity in JavaScript. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2015. (Cited on page 29)

[Pimentel and Nickerson 2012] V. Pimentel and B. G. Nickerson. Communicating and displaying real-time data with WebSocket. *IEEE Internet Computing* (2012). (Cited on page 46)

[Pittappily 2013] T. Pittappily. Semantic zoom. U.S. patent 2013/0067398 A1. 2013. (Cited on page 43)

[Saboury et al. 2017] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. An empirical study of code smells in JavaScript projects. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017. (Cited on page 3)

[Sochat 2019] V. Sochat. Singularity compose: orchestration for singularity instances. *Journal of Open Source Software* (2019). (Cited on page 33)

[You 2014] E. You. *Vue.js*. Vue.js. 2014. URL: `https://vuejs.org/` (visited on 05/01/2020). (Cited on pages 1, 9)