Architectural Consequences of Implementing a Functional Module in Desktop- versus Web-Based Clients

Cebastian Lundbladh, Samuel Hagström

Department of Electrical and Information Technology
Lund University

Supervisor: Christin Swahn

Examiner: Christian Nyberg

June 10, 2026

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This thesis explores the transition from a monolithic desktop application to a distributed web-based client within an industrial software system developed in collaboration with Energy Opticon. The thesis focuses on the implementation and comparison of the Smart Data View (SDV) module in two different client architectures: an existing desktop application built in VB.NET using Windows Forms and DevExpress components, and a newly developed web client built with React, TypeScript, and Fluent UI.

The main objective is to analyze how architectural differences influence system structure, data flow, state handling, and development workflow when implementing the same functionality. The SDV module was implemented in the web client using existing backend APIs, without modifying backend logic or introducing database-level changes.

The analysis is based on key architectural concepts such as layered architecture, event-driven programming, observer patterns, and unidirectional data flow. The desktop implementation relies on a tightly coupled, event-driven and observer-based structure, where UI and logic are closely integrated. In contrast, the web implementation follows a component-based architecture with a clear separation between presentation, state management, and server communication.

The results show that the web architecture improves modularity, separation of concerns, and long-term maintainability, while also introducing more complexity in state management and asynchronous data handling. The desktop architecture offers a more direct and tightly integrated development model, but with reduced flexibility for scaling and evolving the system.

Overall, the thesis highlights the trade-offs involved in migrating from a monolithic desktop system to a modern web-based architecture, particularly in terms of structure, data handling, and development workflow.

Keywords: Software architecture Desktop applications Web applications React State management Component-based architecture Data flow System migration

Sammanfattning

Detta examensarbete undersöker övergången från en monolitisk skrivbordsapplikation till en distribuerad webbaserad klient inom ett industriellt mjukvarusystem som utvecklats i samarbete med Energy Opticon. Examensarbetet fokuserar på implementering och jämförelse av modulen Smart Data View (SDV) i två olika klientarkitekturer: en befintlig skrivbordsapplikation byggd i VB.NET med Windows Forms och DevExpress-komponenter, samt en nyutvecklad webbklient byggd med React, TypeScript och Fluent UI.

Huvudsyftet är att analysera hur arkitektoniska skillnader påverkar systemets struktur, dataflöde, tillståndshantering och utvecklingsprocess vid implementation av samma funktionalitet. SDV-modulen implementerades i webbklienten med hjälp av befintliga backend-API:er, utan att ändra backend-logik eller göra förändringar på databasnivå.

Analysen baseras på centrala arkitektoniska koncept såsom lagerarkitektur, händelsedriven programmering, observatörsmönster och enkelriktat dataflöde. Implementationen i skrivbordsklienten bygger på en tätt kopplad, händelsedriven och observatörsbaserad struktur där användargränssnitt och logik är nära integrerade. Webbimplementationen följer däremot en komponentbaserad arkitektur med en tydlig separation mellan presentation, tillståndshantering och serverkommunikation.

Resultaten visar att webbarkitekturen förbättrar modularitet, ansvarsfördelning och långsiktig underhållbarhet, men introducerar samtidigt ökad komplexitet, särskilt i hantering av tillstånd och asynkron data. Skrivbordsarkitekturen erbjuder en mer direkt och integrerad utvecklingsmodell, men med begränsad flexibilitet när det gäller skalbarhet och vidareutveckling av systemet.

Sammanfattningsvis belyser examensarbetet de avvägningar som uppstår vid migrering från ett monolitiskt skrivbordssystem till en modern webbaserad arkitektur, särskilt vad gäller struktur, datahantering och utvecklingsprocess.

Nyckelord: Mjukvaruarkitektur; skrivbordsapplikationer; webbapplikationer; React; tillståndshantering; komponentbaserad arkitektur; dataflöde; systemmigrering.

Preface

This thesis was carried out as part of the Bachelor of Science in Engineering programme in Computer Science, in collaboration with Energy Opticon. The work focuses on an ongoing transition from a legacy desktop-based application to a modern web-based client, using the Smart Data View (SDV) module as a practical case study.

The purpose of this work has been to gain a deeper understanding of how different architectural choices affect system design, development processes, and long-term maintainability in a real-world industrial system. By implementing the same functionality in two different technological environments, it has been possible to make a direct comparison of differences in structure, data flow, and separation of responsibilities.

The project has provided valuable experience working with both legacy desktop technologies based on the .NET Framework and modern web technologies such as React and TypeScript. It has also highlighted the importance of clear architectural design and well-defined interfaces, particularly in systems that evolve over time.

We would like to thank Energy Opticon for the opportunity to conduct this work within their system and for their support throughout the project. We would also like to thank our academic supervisor and examiner for their guidance and feedback during the process.

Cebastian Lundbladh
Samuel Hagström

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter introduces the thesis by presenting the background, purpose, objectives, and problem statement of the work. It also motivates the relevance of the thesis and describes how the work was divided between the authors. Together, these sections establish the context for the architectural comparison carried out in the later chapters.

1.1 Background

The thesis is carried out in collaboration with Energy Opticon, a software company that develops systems for energy monitoring and analysis. Energy Opticon has a large number of customers within the energy sector, primarily energy companies and industrial organizations across Europe, that use their software for economic and environmental optimization. They currently use a system built as a desktop application in Visual Basic within the .NET Framework. The system works, but it is outdated. To improve long-term maintainability, the distribution model, and technical flexibility, a gradual transition is being made to a web-based client architecture based on React and TypeScript.

The transition involves an architectural change from a locally executed desktop client with a monolithic architecture and internal layer/module structure to a distributed client-server architecture where frontend and backend are logically separated and communicate through well-defined APIs.

As part of this migration, the module "Smart Data View" is implemented from the existing desktop client into the new web client. This transition may result in potential differences in areas such as system structure, data flow, functional decomposition, and development process.

1.2 Purpose

The purpose of the thesis is to analyze and document the architectural differences between desktop and web clients, and to examine how differences in client architecture affect the implementation of the same functional requirements.

1.3 Objectives

Implement the "Smart Data View" on the web client based on existing backend APIs. Ensure correct integration with the existing backend. Identify the architectural differences between desktop and web-based clients with regard to component structure, data flow, division of responsibilities, and state management.

1.4 Problem Statement

The thesis aims to answer the following questions:

1. How is “Smart Data View” structured in the existing desktop architecture?
2. How is the corresponding functionality implemented in a web-based client?
3. Which analysis criteria are most relevant for comparing system structure, data flow, and functional decomposition between desktop and web architectures, and how do these differences influence design decisions?
4. What architectural consequences arise when transitioning from a monolithic desktop client to a distributed web architecture?

1.5 Scope and Delimitations

This thesis is limited to the architectural analysis and implementation of the Smart Data View (SDV) module within Energy Opticon’s client systems. The study compares the desktop and web implementations from a frontend architectural perspective and does not include modifications to backend services, database design, or infrastructure.

The implementation in the web client focuses on the core functionality required for architectural comparison rather than complete feature parity with the desktop version. Some planned features were therefore excluded due to time constraints.

Furthermore, the evaluation is qualitative and based on architectural structure, implementation observations, and development experience. Quantitative measurements such as performance benchmarks, memory usage, or user testing are outside the scope of this thesis.

1.6 Justification for the Thesis

This thesis was chosen because it provides extensive practical experience of how work functions in a professional environment.

For Energy Opticon, the thesis results in a concrete module that can serve as a foundation for continued migration and development. For society and end users, it may contribute to more efficient and user-friendly digital solutions for energy monitoring and analysis.

1.7 Division of Labour

This chapter presents the distribution of work between the authors. The division reflects each author's contribution to the main parts of the project.

Table 1.1: Distribution of work between the authors

	Cebastian Lundbladh	Samuel Hagström
Analysis	80%	20%
Design	40%	60%
Implementation	40%	60%
Testing and Evaluation	30%	70%
Report and Presentation	60%	40%
Poster	50%	50%

Chapter 2
Technical Background

This chapter presents the technical and architectural background needed to understand the comparison between the desktop and web-based implementations. It introduces the main technologies used in the existing desktop client and the new web client, followed by key architectural concepts relevant to the analysis of structure, data flow and state management.

2.1 Desktop Technologies

This section introduces the main desktop technologies relevant to the Energy Opticon desktop client. It provides background on the .NET Framework, Windows Forms, DevExpress, and the architectural concepts needed to understand the existing Smart Data View implementation.

2.1.1 .NET Framework

The .NET Framework is a software development framework created by Microsoft which includes a runtime environment aswell as libraries and tools for developing applications on Windows operating systems. The framework supports multiple coding languages as e.g C# and VB.NET [1].

The framework has two main parts: the Common Language Runtime (CLR) and the class library. The CLR handles memory, threads, and code execution, ensures type safety, and automatically manages objects to prevent errors like memory leaks.

The class library provides ready-made, reusable components for building applications, including tools for graphical interfaces (Windows Forms), web apps (ASP.NET), data handling, and file access [2].

For the Energy Opticon desktop client, the .NET Framework provides the base for building the VB.NET Windows Forms application and implementing modules like the Smart Data View (SDV). VB.NET is a programming language in the Visual Basic family that runs on the .NET Framework and is used to implement application logic and user interface components in the desktop client.

2.1.2 Windows Forms and DevExpress

Windows Forms is a user interface framework in the .NET platform for building desktop applications for Windows. It provides a development model where the user interface is constructed from forms and controls such as buttons, text boxes, menus, and tables. User interactions in these controls generate events that the application handles through code [3].

A central characteristic of Windows Forms is its control-based and event-driven model. A form acts as the visual container for interface elements, while controls are added to display data or receive user input. This makes the framework well suited for traditional desktop applications, where interaction is centered around a persistent graphical interface running locally on the client machine [3].

DevExpress is a third-party component suite that extends Windows Forms with a broader collection of advanced user interface controls and supporting libraries. Its WinForms offering includes controls for grids and editors, charts and other data-visualization components, navigation and layout elements, reporting tools, and features such as theming and accessibility support. By using DevExpress together with Windows Forms, developers can build more feature-rich and data-intensive desktop interfaces than with the standard control set alone [4].

2.1.3 Monolithic Desktop Application Architecture

A monolithic architecture is a software development model that uses a single code base to perform multiple functions, where components are tightly coupled and interdependen [5]. In desktop applications, this typically manifests as a single executable where all functionality is packaged and deployed as one unit. Modules are loaded within the same process and share a common runtime state.

2.1.4 Event-Driven Programming Model

An event is an action that can be responded to in code. Events are typically triggered by user actions, such as keyboard input or mouse clicks, but can also be generated by the system or application logic. Event-driven applications execute code in response to predefined events through event handlers. This event model uses delegates to bind events to their corresponding handler methods. Delegates are type-safe references to methods and their associated target objects, enabling indirect method invocation. They are conceptually similar to function pointers but are object-oriented and type-safe. Delegates define the required method signature for event handlers, ensuring that only methods with a compatible structure can be attached to an event. This provides type safety, as the runtime enforces that invoked methods match the expected event signature. In addition, delegates support multicast behavior, allowing multiple event handlers to be registered and executed sequentially when a single event is raised. In Windows Forms, this event-driven model is implemented through user interface controls such as buttons, text fields, and selection components. Each control exposes a set of predefined events that allow application logic to react to user interactions. Event handlers are attached to these events either at design time or runtime and are executed when the corresponding event is raised. When an event is triggered, the control acts as

the event publisher and invokes the associated delegate, which in turn executes all subscribed event handlers. This mechanism ensures a clear separation between the user interface and the application logic, as controls do not need to be aware of the methods handling their events [6].

2.1.5 State Management in Desktop Applications

In desktop applications, state is often managed locally within individual forms or views rather than through a separate, centralized state management layer. In desktop applications, state is often closely tied to the user interface and is typically maintained by the view that owns the controls and interaction logic. This makes state handling more direct, but also means that state management becomes embedded in the presentation layer rather than isolated as an independent concern [3, 28].

A first category is **UI state**, which consists of values stored directly in form fields, control properties, and other view-specific objects. This may include the current contents of input controls, whether a control is enabled or visible, and temporary values needed to manage the current interaction. Because this state is held locally inside the form, it is usually updated directly in response to user actions and events [3, 30, 31].

A second category is **view state**, which represents the state of a particular view or module. This includes user selections, active filters, presentation settings, and loaded data associated with the current screen. In practice, view state defines the current context of the view and determines what data is shown and how it is presented. In desktop applications, this state is usually encapsulated within the same view that handles the corresponding user interactions.

A third category is **session state**, which concerns preserving and restoring the user's current context between application sessions or when revisiting a view. In Windows Forms, application and user settings can be stored on the client computer and persisted between application sessions [28, 29]. In the desktop client examined in this thesis, this is handled through a memento-based mechanism. In practice, this means that the state of a view can be captured as a snapshot at a given point in time, including selections, filters, and other presentation-related settings, and later restored when the view is reopened. This allows the system to reconstruct the previous context without requiring the user to manually reconfigure the view each time [32].

Another important characteristic of desktop state management is the **absence of a global state manager**. Rather than storing application state in a centralized shared store, desktop applications often keep state locally within individual forms and views. Each view is therefore responsible for maintaining and updating its own state based on the controls and interactions it manages. This results in a simpler and more direct structure, since state is handled close to where it is used. At the same time, it also makes state more tightly coupled to the user interface and more difficult to share, synchronize, or reason about across multiple parts of the system.

2.2 Web Technologies

This section introduces the main web technologies used in the Energy Opticon web client. It provides background on React, Fluent UI, client-side rendering, component-based architecture, state management, and API communication, with focus on how these technologies support the web implementation of the Smart Data View module.

2.2.1 React

React was developed in 2011 and later released to the public in 2013 by Jordan Walke at Facebook [7]. It is a JavaScript library designed for the construction of user interfaces. Primarily used in the development of web applications for browsers, it may also be used in the creation of mobile and desktop applications through the use of additional frameworks and tools [8].

At its core, React is built around the idea that components are self-contained units, meaning each one bundles its own UI and behavior so it can work on its own, like a button, a table, or a form [9]. Instead of using structure, logic, and styling as separate concerns spread across a project, each component sums up all three, making the codebase considerably easier to organize and maintain over time [10].

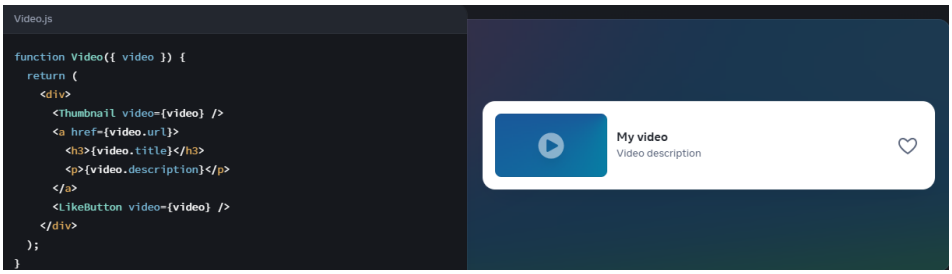

Figure 2.1: Example of React's component-based structure, where a user interface is composed of smaller reusable components [10].

React also uses a Virtual DOM, "Document Object Model". The DOM is the structure that represents the content of a web page. Instead of directly updating the actual DOM whenever a change occurs, React first creates a virtual representation of the interface in memory. When data changes, React compares the new virtual version with the previous one and only updates the parts of the actual DOM that need to be changed. This makes the rendering more efficient and improves performance, especially in more complex user interfaces [11].

Instead of manually controlling each step of how each step of the interface should be updated, React uses declarative programming where the developer simply describes what it should look like given the current state of the application. When data changes, React handles the updating internally, ensuring that the interface always reflects what is actually happening underneath. The result is code that behaves more predictably and is much easier to manage [12].

React Hooks are functions that enable the use of state management and lifecycle-related features within functional components. They were introduced to provide a more short and clear structured approach to component development, reducing the need for class-based components. Some hooks allow developers to handle dynamic data or side effects in a clear and efficient manner, leading to overall improved code, readability and reusability [13].

This component-based structure naturally encourages both modularity and reusability. Dividing the application into separate units meaning that each part can be developed, tested, and maintained independently without unintended side effects spreading throughout the rest of the codebase. Components can also be reused across different parts of the application, reducing repetition and keeping the overall structure consistent. In larger systems this kind of structure becomes necessary, as both efficiency and long-term maintainability depend on a well-organized foundation [9].

2.2.2 FluentUI

Fluent UI is Microsoft's design system and component library for building consistent user interfaces across applications. It provides reusable interface components and design guidance intended to support accessibility, internationalization, and performance in modern user interfaces. For web development, Fluent UI is available for React, making it suitable for applications that require a structured and standardized presentation layer [14].

In React applications, Fluent UI provides prebuilt components and utilities that can be used to construct interface elements such as buttons, inputs, tables, navigation elements, and layout structures. This supports a component-based development model in which interface elements can be reused and composed into larger views while maintaining a consistent visual language [15].

In the context of this thesis, Fluent UI is relevant because it forms part of the presentation layer in the web client together with React. While React provides the component model and rendering logic, Fluent UI provides the visual building blocks used to construct the user interface. This makes Fluent UI important for understanding how the web client is structured and how presentation responsibilities are separated from state management and data access.

2.2.3 Client-Side Rendering Model

Client-side rendering (CSR) is a rendering model in which HTML content is generated in the browser using JavaScript rather than being fully rendered on the server before being sent to the client. In React applications, this means that the browser loads the application code and renders the user interface inside a DOM node in the page [22], [23].

Rendering and updates take place on the client side as application state changes. When state is updated, React determines what parts of the interface need to change and updates the rendered output in the browser accordingly [23], [24]. This makes client-side rendering well suited for interactive web applications where the user interface changes dynamically without requiring a full page reload [22].

In the web client, the browser acts as the runtime environment for the presentation layer, while rendering is driven by JavaScript and component state. This means that the user interface is rendered and updated in the browser as the application state changes.

2.2.4 Component-Based Architecture

Component-based architecture is an approach to software design where the user interface is divided into smaller, reusable components with clearly defined responsibilities [16]. In React, these components are combined into a hierarchy, where larger components are composed of smaller child components. This makes it possible to structure the interface as a tree of related parts rather than as one large and tightly coupled view [17].

A central principle in React's component model is composition over inheritance. React recommends composition as the main mechanism for reusing code between components, instead of building complex inheritance hierarchies. Meaning that components are combined by nesting them inside one another and passing data or content through props. This approach improves flexibility and makes it easier to adapt components to different contexts without introducing strong dependencies between them [16].

A related strategy is UI decomposition, where a user interface is broken down into a component hierarchy based on the structure of the mockup or data model. React describes this as identifying the different parts of the interface, arranging them into parent-child relationships, and assigning each component a limited responsibility. If a component becomes too large or takes on too many responsibilities, it can be divided into smaller subcomponents. [17].

In the context of web applications, component-based architecture supports modularity, reusability, and separation of concerns. By organizing the presentation layer into smaller components, developers can implement, test, and maintain individual parts of the interface more independently [18].

In this thesis, this architectural model is relevant because the web client is structured as a component-based system in which the SDV module is divided into reusable UI components, while non-visual responsibilities are separated into hooks, reducers, and API functions.

2.2.5 State Management in Web Applications

In web applications, state management concerns how application data is controlled, updated, and shared in order to keep the user interface consistent with user interactions and application data. Effective state management is important in modern interactive web applications because it supports smooth data flow, consistency, and predictable responses to changes in both the interface and the underlying data [36, 27, 20].

In React-based applications, state is typically divided into different categories depending on its scope and purpose. A useful distinction can be made between local UI state, shared UI state, and server state, since these types of state have

different requirements and are often managed using different mechanisms [36, 27, 20].

A first category is **local state**, which refers to state that is only relevant to a single component or a small part of the interface. In React, this is commonly managed with the `useState` Hook, which allows a component to store and update values between renders [27]. Local state is well suited for UI-only concerns such as input values, expanded panels, temporary selections, and other interaction-related values that do not need to be shared across multiple parts of the application [27].

A second category is **shared state**, which is needed when multiple components depend on the same data or when state updates must be coordinated across a larger part of the interface. React supports this through mechanisms such as context and reducers. Context allows data to be made available deep in the component tree without passing it explicitly through each intermediate component, while reducers make it possible to centralize state transition logic by describing how state changes in response to actions [33, 34]. This corresponds to the role of shared or global state discussed in the IEEE review, where React's Context API is described as a built-in way to share state across components without prop drilling [36]. When combined, these mechanisms support cross-component coordination and make it easier to manage more complex screens in a structured and predictable way [34].

A third category is **server state**, which refers to data that originates from an external backend rather than being owned entirely by the frontend application. Unlike local UI state, server state must be fetched, cached, synchronized, and updated over time, and it may change independently of the user interface. React Query is designed specifically for this type of state and provides support for asynchronous fetching, caching, background refetching, request deduplication, and handling of loading and error states [20]. React Query also distinguishes server state from local and shared UI state. The difference is that local and shared UI state is created and controlled inside the frontend, for example selected filters, expanded panels, or input values. Server state, however, originates from the backend and can change independently of the user interface. Because the frontend does not own this data, it must be fetched, cached, synchronized, and refetched when relevant parameters change. [20, 35].

This distinction is important in the context of this thesis, because the web client separates UI-related state from backend-driven state. Local and shared interface state are managed through React mechanisms such as reducers and context, while server state is handled through React Query. This creates a more explicit and distributed state model, where different tools are used depending on the type of state being managed.

2.2.6 Data Fetching and API Communication

In web applications, communication with backend systems is typically carried out through HTTP-based APIs. Axios is a promise-based HTTP client for the browser and Node.js, and it is commonly used to send asynchronous requests and receive responses from backend services. According to the Axios documentation, it supports features such as request and response interception, data transformation, timeouts and automatic JSON response handling [19].

TanStack Query, also known as React Query, is a library for managing asynchronous data and server state in React applications. It is used to handle data fetching, caching, synchronization, and updating server state in a more structured and predictable way. Instead of manually managing loading states, errors, and repeated API requests inside components, React Query provides hooks that organize this logic in a reusable manner [20].

In React applications, data can be fetched directly inside components, for example through effects, but React's own documentation notes that frameworks and specialized data-fetching solutions are often more efficient than writing such logic manually in every component [21]. In this context, React Query offers a more structured approach by separating server-related data handling from presentation logic [20].

A central idea in React Query is the distinction between server state and local client state. Server state refers to data that originates from an external backend and therefore needs to be fetched, cached, synchronized and sometimes refetched when conditions change. React Query supports this through features such as automatic caching, background refetching, request deduplication, and built-in handling of loading and error states. This makes it particularly useful in applications where the user interface depends on data from APIs that may change over time [20].

2.3 Architectural Concepts Relevant to Comparison

This section introduces architectural concepts used to compare the desktop and web implementations. These concepts provide a basis for analyzing differences in structure, data flow, state management, and separation of responsibilities.

2.3.1 Layered Architecture

Layered architecture is a commonly used software architectural pattern in which a system is organized into a set of hierarchical layers, each with a distinct responsibility. The primary goal of this structure is to achieve separation of concerns by isolating different aspects of the application, such as user interface, business logic and data access. In a typical layered system, the presentation layer handles user interaction, the application or business layer processes logic and enforces rules and the data access layer is responsible for communication with external systems or databases. Each layer interacts primarily with adjacent layers through well-defined interfaces, which reduces dependencies and improves modularity. This architectural style is widely adopted because it simplifies development and maintenance. By separating responsibilities into layers, changes in one part of the system can often be made without affecting other layers, provided that the interfaces between them remain stable. This also facilitates testing, as individual layers can be developed and verified independently [25].

2.3.2 Unidirectional Data Flow

React is based on a unidirectional data flow model, where data moves in a single direction through the component hierarchy. State is typically managed in higher-

level components and passed down to child components through properties (props). User interactions in child components do not directly modify shared state, but instead trigger callbacks that update the state in the parent component.

This design follows the principle that the user interface is a function of the application state. When state changes, React re-renders the affected components to reflect the updated state. As described in the React documentation, applications should be structured by first identifying the minimal state required and then deriving the user interface from that state representation [17].

This approach contrasts with desktop application architectures, where user interface logic and application state are often more closely coupled and updated through event-driven mechanisms.

2.3.3 Observer Pattern

The Observer pattern is a behavioral design pattern where one object (the publisher) keeps a list of dependent objects (subscribers) and automatically notifies them when its state changes. Instead of other objects constantly checking for updates, they are informed through a notification mechanism.

The pattern works by letting subscribers register themselves with a publisher. The publisher stores these subscribers and calls a common update method on them whenever an event or state change occurs. Because all subscribers follow the same interface, the publisher can communicate with them without knowing their specific implementation.

This allows multiple parts of a system to react to the same change while remaining loosely connected. New subscribers can be added or removed at runtime without changing the publisher.

In user interface systems, this pattern is commonly used to update UI components when data or application state changes [26].

2.3.4 Server State vs Client State

In web applications, application state can be divided into two main categories: client state and server state. The distinction is based on where the data originates and how it is managed within the application.

Client state refers to data that exists within the frontend application and is primarily related to user interface behavior. This includes temporary values such as form inputs, selected filters, toggles, and other interaction-related data. In React, this type of state is managed within components using built-in state mechanisms such as `useState`, which allows components to store and update local state during runtime [27].

Server state refers to data that originates from an external backend system, typically accessed through APIs. This includes data stored in databases or services that can change independently of the frontend application. Server state must be fetched, cached, synchronized, and updated based on communication with the backend, since it is not owned by the client.

In React-based applications, server state is often managed using specialized libraries such as React Query. React Query provides tools for handling asyn-

chronous data fetching, caching, background updates, and synchronization with server data. It also manages common concerns such as loading and error states, reducing the need for manual implementation within components [20].

2.4 Terminology

2.4.1 Smart Data View (SDV)

The Smart Data View (SDV) module in the desktop client is implemented as a WinForms view using DevExpress components, known internally as `MixedDataView`. It orchestrates multiple elements, including lookup functionality, filtering criteria, charts, and data grids [3], [4].

For the purpose of this thesis, the SDV module refers to this functionality and its corresponding implementation in the web client.

2.4.2 Client-side rendering (CSR)

Client-side rendering (CSR) is a rendering model where the user interface is generated and updated in the browser using JavaScript [22]. In the context of this thesis, CSR is relevant because the web client renders the SDV interface in the browser rather than receiving a fully rendered page from the server.

2.4.3 Document Object Model (DOM)

The Document Object Model (DOM) is the browser's representation of a web page structure. React interacts with the DOM by updating the parts of the interface that need to change when application state is updated [11], [24].

2.4.4 Hooks

Hooks are functions in React that allow functional components to use features such as state, side effects, and reusable logic [13], [21]. In this thesis, hooks are relevant because the web implementation uses custom hooks to separate data fetching and application logic from presentation components.

2.4.5 Reducer

A reducer is a function used to update state based on defined actions. In the web implementation, reducers are used to handle structured state transitions and make updates to the user interface state more predictable [34].

2.4.6 Props

Props are values passed from a parent React component to a child component. They are used to transfer data, settings, or callback functions between components in the component hierarchy [9], [17].

2.4.7 React Query

React Query is a library used to manage server state in React applications. It handles data fetching, caching, loading states, error states, and synchronization with backend APIs [20], [35].

2.4.8 Axios

Axios is a JavaScript HTTP client used to send requests to backend APIs and receive responses. In the web implementation, Axios is used in the data access layer to handle HTTP communication [19].

2.4.9 Azure DevOps

Azure DevOps is a platform used for project planning, version control, pull requests, and sprint management. In this thesis, it was used to manage development tasks, branches, code reviews, and integration into the shared repository [37].

Chapter 3
Method

This chapter describes the method used to plan, implement, and analyze the SDV module. It presents the workflow of the thesis, the implementation process, the analytical framework used for comparison, and how sources and AI-based tools were evaluated.

3.1 Thesis Planning

The thesis work was divided into four phases to systematically conduct the implementation and analysis:

1. **Thesis Planning** - Included task distribution, sprint planning for the implementation, and defining criteria for architectural analysis.
2. **Information Gathering** - Studying desktop and web architectures as well as API documentation. This phase was conducted in parallel with the other phases, as information was needed continuously throughout the work of the thesis.
3. **Implementation of the SDV Module** - Work was divided into small tasks, each implemented in a separate branch, submitted as pull requests, and reviewed to ensure code quality and consistency with the rest of the client.
4. **Architectural Comparison** - A detailed analysis of architectural differences between the desktop and web clients, focusing on component structure, data flow, division of responsibilities, and state management.

Phases 1, 3, and 4 were carried out sequentially, while phase 2 (information gathering) was performed in parallel with the other phases as shown in Figure [3.1].

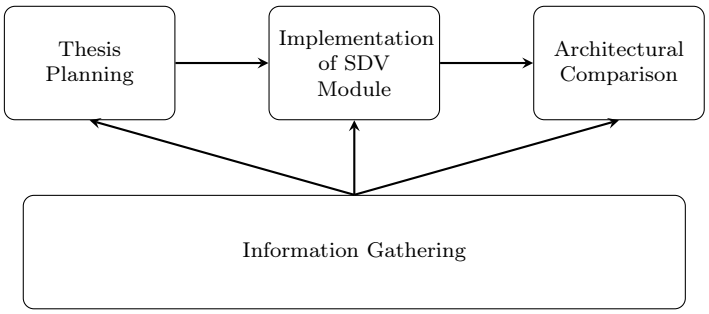

Figure 3.1: Flowchart of the workflow.

3.1.1 Analytical Framework

When comparing the desktop and web-based implementations of the SDV module, a set of analytical criteria was defined. These criteria focus on the core architectural aspects that influence how functionality is structured, implemented, and maintained.

The following areas were selected for the analysis:

- **Component Structure and Functional Decomposition** - Examines how the system is divided into components, modules, and layers, as well as how functionality and responsibilities are distributed between them. This includes how reusable, modular, and clearly separated the different parts of the implementation are.
- **Data Flow** - Describes how data moves through the system, including how it is retrieved, transformed, and passed between different parts of the application.
- **State Management** - Investigates how application state is stored, updated, and synchronized, particularly in relation to user interactions and data updates.

These criteria were chosen because they capture key aspects of software architecture affected by the transition from a desktop application to a web-based client. Together, they provide a clear basis for identifying architectural differences and understanding their impact on design decisions, system behavior, and maintainability.

3.2 Information Gathering

Data for the analysis was collected from multiple sources to ensure a comprehensive understanding of both the desktop and web-based implementations of the SDV module.

The existing desktop application was examined through code inspection, focusing on the structure, components, and internal data handling of the SDV module.

A literature study was conducted to gain a theoretical understanding of relevant architectural concepts such as layered architecture, event-driven systems, and modern frontend design patterns. This provided a foundation for the analysis and comparison between the two implementations.

The web-based implementation was analyzed during development, providing insight into component structure, data flow, and integration with backend APIs.

Github Copilot were also used to inspect and better understand data flows within the system. These tools supported the identification of patterns and relationships in the code, but all observations and conclusions were verified through manual analysis.

Finally, functional testing and observations were conducted to verify system behavior and to understand how the module operates in practice.

3.3 Implementation of the SDV Module

The implementation work was carried out both on-site and remotely. The work arrangement included three days per week in the office, while the remaining days were remote. Communication with the frontend developers and other team members was conducted through Microsoft Teams to clarify questions and request guidance when needed.

The work was organized using a hierarchical structure of epics, features, user stories, and tasks as shown in Figure [3.2]. Each user story was broken down into separate tasks, which were placed on the sprint board until the next sprint review. Sprints occurred every second Wednesday.

For development, the existing Git repository was cloned and a separate branch was created for each user story assigned to us. Upon completion, a pull request (PR) was submitted on Azure DevOps, which manages the sprint board, backlog, and repository. PRs were reviewed, comments addressed, and then approved and merged into the parent branch. Tasks were assigned such that multiple tasks could be worked on in parallel.

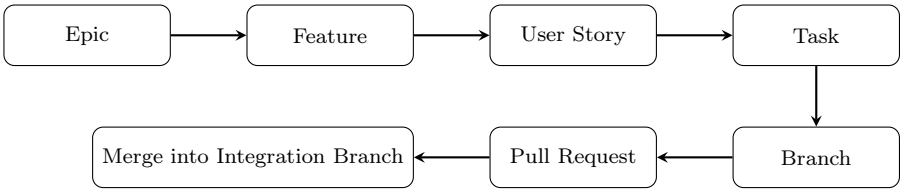

Figure 3.2: Development workflow from epic to merge in the Git repository.

3.3.1 Backend Integration

The SDV module in the web client does not communicate directly with the backend from the UI components. Instead, it uses a structured approach where data is

fetched through reusable React hooks. These hooks act as an intermediate layer between the UI and the backend APIs.

The hooks call API functions that handle the actual HTTP communication using Axios. Configuration such as API base URLs and authentication is centralized, meaning all requests automatically include the required authorization.

Asynchronous data fetching is handled using React Query. This allows the application to cache results, avoid duplicate requests, and manage loading and error states in a consistent way. Requests are only sent when valid input data is available, preventing unnecessary API calls.

Since SDV depends on multiple types of data, results from different hooks are combined into a unified structure before being used in the UI. The main component updates its state only after data has been successfully fetched, which helps avoid inconsistent or partial data being displayed.

To keep the UI in sync with the backend, data is refetched when important parameters change, such as user-selected filters or application language.

This is further visualized in Figure [3.3].

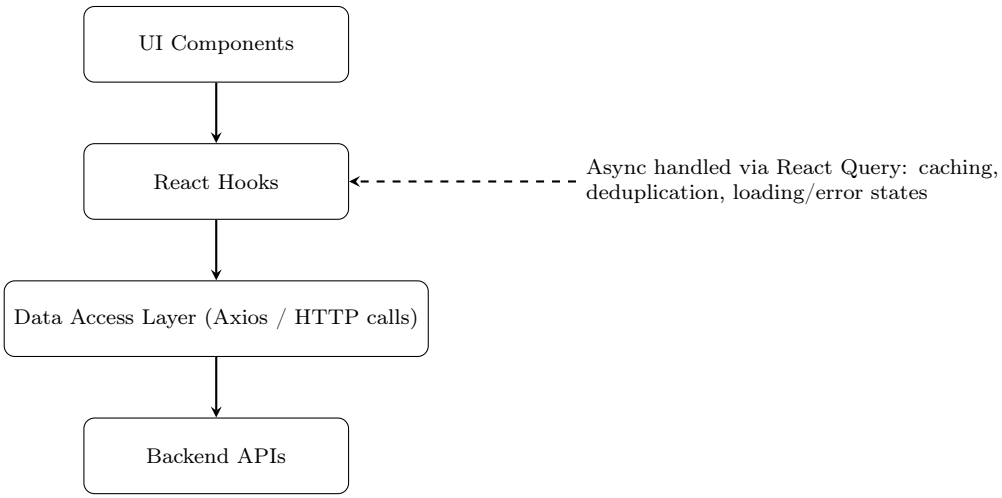

Figure 3.3: Backend integration and data flow in the SDV web client.

3.4 Architectural Comparison

The comparison was conducted by comparing the desktop and web-based implementations of the SDV module using the analytical framework defined earlier.

The desktop implementation was analyzed through code inspection supported by Github Copilot, which were used to navigate the codebase, identify relevant components, and trace data flows. This approach enabled efficient exploration of a large and complex system. Observations were validated by examining selected parts of the code and by relating findings to observed system behavior.

In contrast, the web-based implementation was evaluated based on direct development experience, as the SDV module was implemented as part of this thesis. This provided a detailed understanding of component structure, state management, and data flow, as well as the reasoning behind architectural decisions.

A mapping process was then carried out to align corresponding functionality between the desktop and web implementations. This enabled a structured comparison of how the same functional requirements were realized in each architecture.

Finally, the differences were analyzed using the defined criteria: component structure and functional decomposition, data flow, and state management. The impact of these differences was evaluated in terms of maintainability, modularity, complexity, and system transparency.

3.5 Use of AI in the Thesis

We confirm that this Bachelor's thesis was written by us and that the work presented is our own, in terms of analysis, structure, and final responsibility, except where explicitly stated otherwise. This thesis has not been submitted for any other degree or professional qualification, nor has it been published previously, unless otherwise specified. All sources used, whether printed, online, or otherwise, have been properly acknowledged and referenced.

During the preparation of this thesis, we used ChatGPT and GitHub Copilot to assist with improving language, structure, and readability, as well as to provide feedback on written code. All content generated with the assistance of these tools has been carefully reviewed, edited, and validated by us, and we take full responsibility for the final content of this thesis.

3.6 Source Evaluation

The technical background in this thesis is based on information gathered from a range of official documentation, educational material, industry sources, and academic literature. When using sources, their credibility, purpose, and scientific validity must be considered.

During the literature study, most sources were found to be technically correct and well-structured, particularly official documentation from technology providers such as Microsoft [2, 3, 6, 28, 29, 30, 31], React [9, 10, 11, 12, 16, 17, 21, 23, 24, 27, 33, 34], Axios [19], TanStack Query [20, 35], and Fluent UI [14, 15]. These sources are considered highly reliable since they are maintained by the organizations responsible for the technologies themselves. They are therefore used as primary references for framework behavior, architecture, and implementation details.

3.6.1 Official documentation

Official documentation forms the most reliable category of sources in this thesis. Microsoft Learn documentation for the .NET Framework and Windows Forms [2, 3, 6, 28, 29, 30, 31] provides authoritative descriptions of the desktop architecture,

including event-driven programming and UI structure. DevExpress documentation [4] is used to describe extended UI components used in the desktop client.

React documentation [9, 10, 11, 12, 16, 17, 21, 23, 24, 27, 33, 34] is used to describe core concepts such as component structure, rendering behavior, hooks, and unidirectional data flow. These sources are considered primary references because they define the frameworks directly.

Fluent UI documentation [14, 15] is used to describe the UI component system used in the web client.

Axios [19] and TanStack Query [20, 35] are used to describe API communication and server-state management in the web application.

In addition, general web platform concepts such as client-side rendering are described using documentation from Mozilla Developer Network [22], which is considered an authoritative reference for browser-related technologies.

3.6.2 Industry sources

Industry sources such as Amazon Web Services (AWS) and Refactoring Guru are used to support architectural and design concepts.

AWS documentation on monolithic and microservices architectures [5] is considered reliable due to its position as a major cloud provider and is used to describe architectural differences between system types.

Refactoring Guru [26, 32] is used to explain the Observer design pattern aswell as the memento pattern. Although not an academic source, it is widely used in software engineering education and aligns with established theoretical definitions.

3.6.3 Educational and online sources

Educational sources such as GeeksforGeeks [1, 18], W3Schools [7], Epic React [8], and LoginRadius [13] are used for introductory explanations of technical concepts.

These sources are useful for providing simplified and accessible descriptions but are not considered fully authoritative, as they are not peer-reviewed and may vary in technical depth. Therefore, they are used as supporting material alongside official documentation.

3.6.4 Academic literature

Academic literature, such as *Fundamentals of Software Architecture* by Richards and Ford [25] and [36] by Dron, M. and Szandała, T., provides theoretical grounding for architectural analysis. These sources are considered highly reliable due to academic and professional publishing standards and are used to support discussions on system design, state management, layering, and architectural trade-offs.

3.6.5 Summary

Overall, the thesis prioritizes official documentation and academic literature due to their high reliability and direct connection to the technologies and concepts used. Industry sources are used to support architectural reasoning, while educational sources are used to improve accessibility and understanding of complex topics.

Chapter 4
Analysis

This chapter analyzes the desktop and web-based implementations of the SDV module. The analysis focuses on architectural structure, component responsibilities, data flow, state management, and the consequences these differences have for maintainability and system design.

4.1 Overview of Desktop Architecture

The desktop client is implemented as a modular Windows application where functionality is divided into distinct modules within a single monolithic executable. The system follows a layered architecture consisting of a presentation layer built on Windows Forms and DevExpress components, a data access that interfaces with external backend APIs.

While the desktop client does not define a dedicated state management layer, state is handled within individual views, effectively embedding state management within the presentation layer.

The architecture is centered on a single application entry point, which initializes the system, configures dependency injection to manage module dependencies, and manages module navigation through a shared application framework. Each module is loaded into this shell as an independent view, where it is responsible for composing its own layout, handling user interactions, and coordinating data retrieval through the data access layer.

State management is distributed across individual views and components. Each view maintains its own local state, including user selections, filter configurations, and loaded data. State can be persisted and restored using a memento-based mechanism, which serializes the current view context, meaning it converts the current state into a storable format and deserializes it on reload, allowing the system to preserve user context between sessions.

Data handling is abstracted through the data access layer, which exposes defined interfaces to the presentation layer. The UI interacts exclusively with these interfaces rather than communicating directly with backend APIs. Internally, the data access layer delegates to contributor-based implementations that are resolved at runtime, each responsible for retrieving, transforming, and aggregating data from a specific external API or data source.

The system follows an event-driven and command-based interaction model.

User actions trigger events and commands that propagate through the system, updating view state or initiating data retrieval through the data access layer. Data sources notify subscribed components when new data becomes available via an observer pattern, which in turn triggers re-rendering of the relevant UI elements.

Overall, the desktop client can be characterized as a stateful, event-driven system with a layered monolithic architecture, where responsibilities are separated across layers as visualized in Figure [4.1].

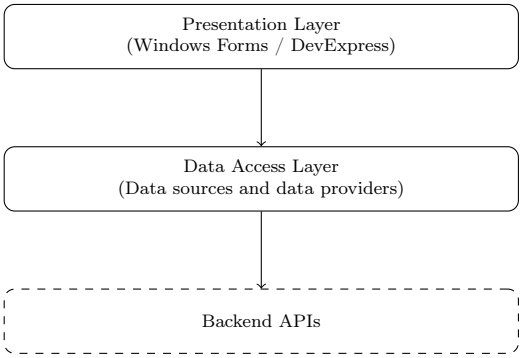

Figure 4.1: Layered architecture of the desktop client.

4.2 Overview of Web Architecture

The SDV module is implemented as a self-contained, modular part of the web client, designed so that its logic is separate from the rest of the system. This makes it easier to develop and maintain since it is self-contained. The system follows a layered architecture consisting of a presentation layer built using React Components and FluentUI, a state management layer (reducers and context), and a data access layer (hooks and API functions), which communicates with backend APIs.

The architecture is centered around a single entry point that initializes the page distributes shared state to the rest of the components. The application is structured as a component-based system where functionality is divided into reusable UI components.

The state itself is divided into clear sections of responsibility, such as data selection, presentation settings and user interface. A reducer governs state transitions, ensuring that updates follow a predictable pattern and that unrelated concerns remain isolated from one another. This separation reduces coupling between components and makes the system easier to extend and test independently.

Data handling is abstracted away from the UI. Instead of components directly managing data retrieval, dedicated logic is responsible for fetching and preparing the data. Asynchronous behaviour, including caching, request deduplication, and loading and error state management, is handled by React Query, preventing redundant network requests and ensuring consistency in how data states are presented.

The system follows a one-way data flow: user actions trigger updates to the state, which leads to new data being fetched and the interface being updated accordingly.

In the SDV web implementation, Fluent UI was used as the main component library for constructing the presentation layer, including interface elements such as buttons, input controls, layout elements, and tables. Its role was mainly visual and structural, while state management and data fetching were handled separately through reducers, hooks, and React Query.

Since the web client uses client-side rendering, these state changes are reflected directly in the browser without requiring a full page reload. This supports the interactive behavior of the SDV module, where filters, tables, and visual components are updated based on changes in application state.

Overall, the web client can be characterized as a state-driven, component-based system with a layered architecture and one-way data flow, where responsibilities are clearly separated between presentation, state management and data access layers.

This is further visualized in Figure [4.2].

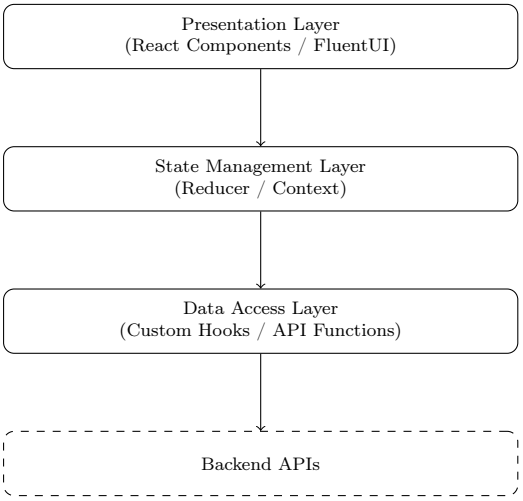

Figure 4.2: Layered architecture of the web client.

4.3 Component Structure and Responsibility Distribution

The SDV module is structured differently in the desktop and web clients, particularly in terms of how functionality is decomposed and how responsibilities are distributed. These differences directly affect modularity, separation of concerns and reusability.

4.3.1 Desktop component structure and responsibilities

In the desktop client, the SDV module is implemented as a single integrated view within a Windows Forms-based application. The main view (internally referred to as a MixedDataView) encapsulates multiple responsibilities, including layout composition, user interaction handling, state management, and coordination of data retrieval.

UI elements such as filter controls, data grids, and charts are contained within the same view component. While internal helper classes and data access exist, they are typically accessed and orchestrated through the view itself.

This results in a feature-centric structure where responsibilities are co-located within one module. While this simplifies navigation and reduces structural fragmentation, it also increases internal coupling and makes individual concerns harder to isolate.

4.3.2 Web component structure and responsibilities

In the web client, the SDV module is decomposed into smaller, reusable React components. The interface is split into a container component and multiple child components responsible for specific UI elements such as filters, tables and visualizations.

Non-visual logic is extracted into custom hooks and state management utilities. Data retrieval is handled through dedicated hooks, while reducers manage local UI state transitions. API functions are responsible for direct communication with backend APIs.

This separation creates a clear division of concerns between presentation, state management, and data access. Each part of the system has a narrowly defined responsibility, enabling independent development, testing and reuse.

4.3.3 Comparison

The desktop implementation follows a feature-centric model where responsibilities are concentrated within a single cohesive view. In contrast, the web implementation follows a layered and responsibility-centric model, where functionality is distributed across components, hooks, reducers, and API functions.

As a result, the desktop client exhibits higher internal coupling, while the web client achieves greater separation of concerns and improved modularity.

4.4 Data Flow Comparison

The data flow within the SDV module differs significantly between the desktop and web implementations. These differences are primarily related to how data is requested, processed, and propagated through the system, as well as how updates are reflected in the user interface.

4.4.1 Desktop data flow

In the desktop client, data flow follows a layered and event-driven pipeline. User interactions in the SDV view, such as applying filters or changing selections, trigger events or commands within the view layer. These events are propagated to the underlying data access layer, which is responsible for retrieving data from external sources through a Data Access Layer.

Once data is retrieved, it is passed through a series of internal layers where it may be transformed or aggregated before being returned to the presentation layer. The UI components then update based on the received results, often through observer-based notifications or direct view updates.

This results in a data flow that is both layered and bidirectional, where requests travel downward through the architecture and results propagate back upward through multiple abstraction layers as seen in Figure [4.3]

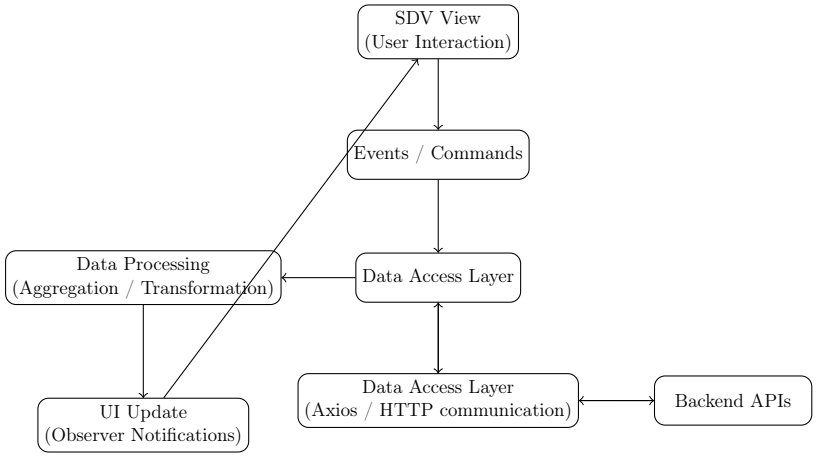

Figure 4.3: Desktop data flow in the SDV module, showing event-driven requests and observer-based UI updates.

4.4.2 Web data flow

In the web client, data flow is structured around a unidirectional model. User interactions in the UI trigger state updates, which are handled either by a reducer (for UI state) or by React Query (for server state). When server-related state changes occur, data-fetching hooks are responsible for initiating API requests to the backend.

The retrieved data is then cached and managed by React Query, which ensures consistency, prevents redundant requests, and handles loading and error states. Once the data is updated, it is passed back to the components through hooks, which automatically trigger re-rendering of the affected UI elements.

This creates a clear data flow as displayed in Figure [4.4]

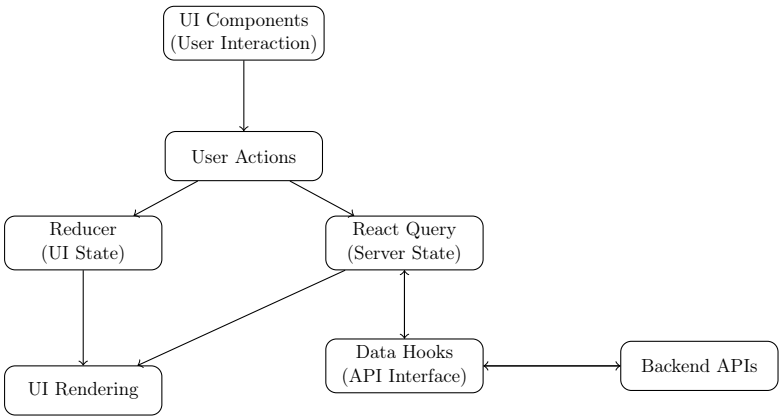

Figure 4.4: High-level web architecture and data flow of the SDV, showing separation between UI state and server state management.

4.4.3 Comparison

The desktop implementation follows a multi-layered and event-driven data flow, where data passes through several intermediate layers before reaching the UI. This introduces additional abstraction but also increases the complexity of tracing data movement through the system.

In contrast, the web implementation uses a more direct and predictable unidirectional data flow. State changes explicitly trigger data fetching and UI updates, reducing the number of intermediary steps involved in data propagation.

4.5 State Management

This section compares how state is managed in the desktop and web implementations of the SDV module, with a focus on how state is stored, updated and shared across the system.

4.5.1 Desktop state management

In the desktop client, state is primarily managed within individual views. Each view maintains its own internal state, typically represented through object fields and properties. This includes UI-related state such as user selections, filter configurations and loaded data.

State is generally encapsulated within the view that owns it, with limited sharing between components. When state needs to be persisted or restored, a memento-based mechanism is used to serialize and reconstruct the view context.

As a result, state management is closely tied to the UI layer and follows an object-oriented approach where state is kept within the specific component that uses it, rather than being shared globally by default.

4.5.2 Web state management

In the web client, state is divided into multiple categories depending on its purpose. Local UI state is managed using React's built-in state mechanisms such as `useState` and reducers. Shared UI state is handled through context or reducer-based state management.

Server-related state is managed separately using React Query, which provides caching, synchronization and background updates. This separates UI state from server state and reduces the need for manual data handling within components.

This results in a more distributed state model where different tools are used depending on the type and scope of the state.

4.5.3 Comparison

The desktop implementation uses a centralized and view-bound state model, where state is tightly coupled to individual UI components. In contrast, the web implementation separates state into local UI state and managed server state, resulting in a more distributed and reactive model.

This makes the web client more flexible in handling asynchronous data, while the desktop client provides a simpler but more tightly coupled state structure.

4.6 Architectural Consequences

The differences between the desktop and web implementations have direct implications for maintainability, extensibility and system transparency.

In the desktop client, functionality is concentrated within large, tightly coupled views. This simplifies the overall structure but increases the risk that changes in one part of the SDV module affect unrelated behaviour. The layered and event-driven design also introduces multiple intermediate steps in data processing, which can make it more difficult to trace how data is transformed and propagated through the system.

In contrast, the web client distributes responsibility across smaller components, hooks and API functions. This improves isolation between concerns, making changes easier to localize and test. The unidirectional data flow also increases predictability, since state updates follow a consistent cycle from user interaction to data fetching and UI rendering.

However, the increased modularity in the web architecture introduces a larger number of interacting parts. This requires stricter conventions for state management and hook design to avoid unnecessary re-renders, redundant data fetching, or fragmented logic spread across the system.

Overall, the desktop architecture favors simplicity and centralization, while the web architecture prioritizes modularity and predictable data flow at the cost of increased structural fragmentation.

Chapter 5
Results

This chapter presents the results of the SDV implementation and the main findings observed during development. It focuses on the implemented functionality, interaction flow, and differences identified between the desktop and web-based clients.

The implementation of the SDV module was carried out to a level sufficient to support the analysis and answer the research questions defined in this thesis. Full feature completion was not a requirement, as the primary objective was to implement and evaluate the core functionality necessary to compare the desktop and web architectures.

The implemented functionality covers the main user flows, including data retrieval, filtering, and visualization, which form the basis for the architectural comparison between the two systems.

5.1 Implementation Findings

The implementation of the SDV module in the web client resulted in a functional version of the core features present in the desktop application. These include data filtering, lookup functionality, tabular representation, and visual components for data presentation.

Integration with backend APIs was achieved through a structured data access layer consisting of custom hooks and API functions. All server-related data handling was performed using React Query. No direct API calls or manual data fetching logic were implemented within UI components. Instead, data access was fully abstracted through reusable hooks.

One of the main implementation challenges was related to combining and transforming data from multiple API endpoints into a unified structure suitable for presentation in the user interface. This was partly due to limited knowledge of the underlying data structures and inconsistencies in how data was represented across endpoints. As a result, additional effort was required to interpret and align the data before it could be used in the SDV components.

The web implementation relies on asynchronous data handling, where data is fetched based on user interactions and application state. During development, the web client was observed to provide responsive updates in the user interface, although no formal performance measurements were conducted. In practice how-

ever, the web-based implementation was perceived as more responsive during interaction compared to the desktop client, particularly when applying filters and updating visual components. This was mainly attributed to the asynchronous rendering model and efficient state-driven updates provided by React and React Query, which reduce blocking UI updates and avoid full view refreshes.

The implemented SDV module covers the majority of the planned functionality required for the study. Core features such as data retrieval, filtering, lookup, and visualization were fully implemented and operational. However, certain components were not completed, including the optimized variables tab, properties view, and direct database integration. In addition, several minor interface refinements and supporting features remain unimplemented. Despite these limitations, the available functionality was sufficient to support the intended architectural comparison.

The incomplete implementation is primarily due to time constaints and the limited scope of the thesis. The work was counducted within a fixed timeframe, which required proritizing core functionality necessary for the architectual comparison over full feature completion. In addition, parts of the system depended on external components and domain-specific knowledge that required additional time to fully understand and integrate.

5.1.1 Visual Representation of the SDV Module

Figures [5.1] and [5.2] illustrate the SDV module in the desktop and web clients, respectively. Figure [5.1] presents the fully developed desktop implementation with all available features, while Figure [5.2] shows the current state of the web-based implementation. The web client includes the core functionality developed in this thesis, such as data tables and filtering components, but does not yet contain all features present in the desktop version.

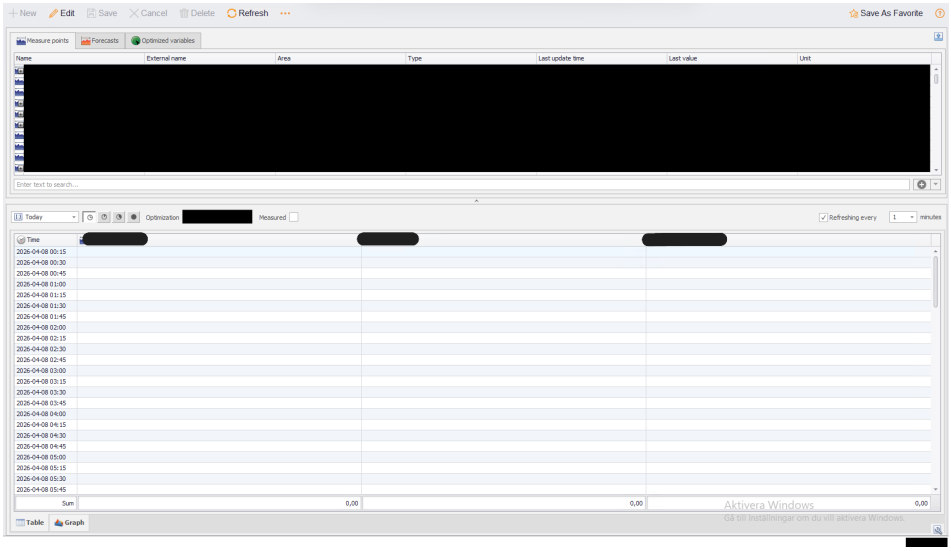

Figure 5.1: Screenshot of SDV in desktop client

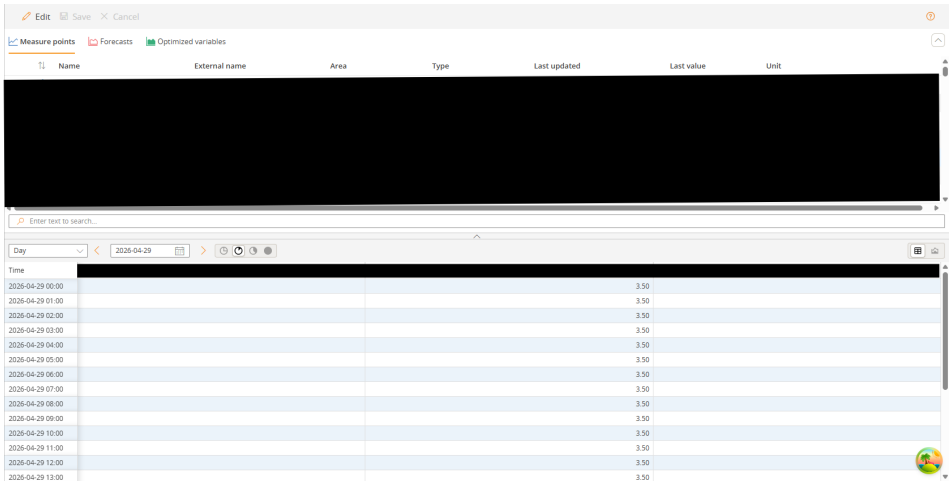

Figure 5.2: Screenshot of SDV in web client

5.2 Observed Interaction Flow

A concrete difference between the desktop and web implementations was observed in how filter interactions are processed and propagated through the system.

In the web client, user interaction with filter controls triggers a state update through a centralized state management mechanism. The updated state modifies the query parameters used for data retrieval, which results in automatic refetching

of data. Once new data is available, the affected components are re-rendered, updating tables and visualizations accordingly.

In the desktop client, filter interactions are handled through a parameter-based mechanism. User selections are combined into a parameter state representing the current criteria, including time selection, resolution, and presentation settings. These parameters are applied to the data source, which updates the bound components such as charts and tables. Updates are propagated through event-driven mechanisms, where the data source notifies the view when new data is available.

5.3 Summary

The results show that the implemented SDV functionality was sufficient to support a meaningful comparison between the desktop and web clients. Even though the module is not fully completed in terms of all planned features, the core functionality was successfully implemented and used as the basis for the study.

Both implementations support similar user-facing functionality, including filtering, data retrieval, and visualization. Differences were observed in how data is handled, how state is updated, and how user interactions propagate through the system.

Chapter 6
Discussion and Conclusion

This thesis has analyzed and compared the architectural differences between a monolithic desktop client and a component-based web client through the implementation of the SDV module. Both implementations provide equivalent functionality, which enabled a direct comparison of how identical requirements are realized under different architectural constraints.

6.1 Discussion

The results of this thesis show that implementing the same functionality in two different client architectures leads to different architectural trade-offs, even when the user-facing requirements are largely the same. The SDV module provides a useful case for this comparison because it includes several aspects that are strongly affected by architectural design, such as filtering, data retrieval, visualization, state handling, and interaction flow.

One important observation is that the desktop implementation provides a more direct and integrated development model. Since state, user interaction logic, and view updates are handled close to the user interface, the flow of control can be simple to follow within a single view. This can be beneficial when implementing smaller changes or when working with functionality that is already well understood. However, this structure also means that many responsibilities are concentrated in the same part of the system. As a result, changes to the view may affect several different concerns, such as layout, data handling, and state management. This increases coupling and can make the system harder to maintain as the module grows.

The web implementation instead distributes responsibility across several layers and smaller units, such as components, hooks, reducers, and API functions. This improves separation of concerns and makes it easier to isolate individual parts of the system. For example, user interface components do not need to contain the details of backend communication, since this responsibility is handled by hooks and API functions. Similarly, server state is managed separately from local user interface state. This structure supports maintainability and makes the system easier to extend, but it also introduces more architectural overhead.

A central trade-off identified during the work is therefore the balance between simplicity and modularity. The desktop client is simpler in the sense that related

functionality is often located in one view, but this simplicity comes with tighter coupling. The web client is more modular and explicit, but requires developers to understand how several separate parts interact. This means that the web architecture is not automatically easier to work with; its advantages depend on consistent conventions, clear naming, and disciplined separation between presentation, state management, and data access.

State management was one of the most significant differences between the two implementations. In the desktop client, state is mainly view-bound and updated through events and direct interaction with controls. This makes state handling straightforward in local scenarios, but less transparent when several parts of the view depend on the same data. In the web client, state is handled more explicitly. UI-related state is separated from server state, and updates follow a more predictable cycle from user interaction to state update, data fetching, and rendering. This improves traceability, especially in asynchronous scenarios, but also requires careful design to avoid fragmented or duplicated state logic.

The data flow comparison also shows how the architectural model affects system transparency. In the desktop client, data can pass through several layers and observer-based update mechanisms before it reaches the user interface. This provides abstraction, but can make the exact flow of data more difficult to trace. In the web client, the flow is more explicit because changes in state trigger data-fetching logic and re-rendering in a structured way. This makes the relationship between user actions, backend communication, and UI updates easier to reason about, especially when working with asynchronous data.

The implementation process also highlighted the importance of understanding the existing data model and the API functions already available in the web client. The SDV implementation did not require new backend endpoints or database changes. Instead, it reused existing API functions that were already used by another module in the web client. Much of the complexity was therefore related to understanding how these existing API functions represented the data, and how the returned data could be combined and transformed into a structure suitable for the SDV interface.

This shows that architectural migration is not only about recreating user interface functionality in a new technology. Even when existing API functions can be reused, the client architecture still affects how data is requested, transformed, stored, and presented in the interface

It is also important to consider the limitations of the comparison. The web implementation did not include all functionality available in the desktop version, and no formal performance measurements were conducted. Therefore, conclusions about performance and complete feature equivalence should be made carefully. The comparison is primarily qualitative and focuses on architectural structure, development experience, and observed behavior. Even so, the implemented core functionality was sufficient to identify important architectural differences between the two clients.

Overall, the discussion shows that the transition from a monolithic desktop client to a web-based client involves more than replacing one technology with another. It changes how responsibilities are divided, how state is represented, how data flows through the system, and how developers reason about functionality.

The web-based approach offers stronger modularity, clearer separation of concerns, and better support for asynchronous data handling, but it also requires more architectural discipline to prevent unnecessary complexity. The desktop approach remains more direct and locally cohesive, but becomes less flexible as the system grows and evolves.

6.2 Research Problems

The following research problems, previously presented in the problem statement section of the introduction, structure the analysis conducted in this thesis:

1. How is "Smart Data View" structured in the existing desktop architecture? The desktop client concentrates functionality within large, tightly coupled views, where state and logic are closely integrated. This results in a simpler local structure but increases internal coupling and makes long-term changes harder to isolate.

2. How is the corresponding functionality implemented in a web-based client? In contrast, the web client distributes responsibility across clearly separated layers, including components, state management, and data access. This improves modularity and makes individual parts of the system easier to reason about and maintain independently. However, it also introduces a higher number of abstractions, which also means the architecture requires more careful structuring to avoid scattered logic across components.

3. Which analysis criteria are most relevant, and how do they influence design decisions? The comparison highlights that data flow and state management are the most significant areas affected by the architectural shift, as they directly determine how system behavior is coordinated and how predictable the application remains under asynchronous updates.

The existing desktop architecture serves as a reference point, where state is primarily managed locally within views and propagated through event-driven mechanisms. This results in a less strictly structured communication model between components, where state changes can be triggered from multiple sources and are not always centrally coordinated.

This observed structure influenced the design decisions in the web architecture, where a more explicit and constrained data flow was introduced. In practice, this led to design choices that separate UI state from server state more clearly and enforce more structured update paths for state changes, typically driven by API responses or controlled state updates rather than direct event propagation between components. This design reduces implicit dependencies between components and increases control over how and when state changes occur.

4. What architectural consequences arise when transitioning to a web architecture? The findings show a fundamental shift in system design when migrating from a desktop to a web-based architecture. Overall, the

transition from a desktop to a web-based architecture represents a shift from centralized implementation toward distributed modularity. This improves scalability and maintainability but introduces additional structural complexity that must be actively managed.

The results demonstrate that architectural decisions have a direct impact on how systems evolve, even when functional requirements remain unchanged.

6.3 Ethical Aspects

The transition from a monolithic desktop architecture to a more modular web-based system introduces several ethical considerations related to software development practices, long-term maintainability, and the use of AI-assisted tools.

As systems become more modular and built on higher-level frameworks, the required skill set for development and maintenance may gradually shift. Tasks that previously required deep knowledge of low-level implementation details are increasingly abstracted away. This can influence how responsibilities are distributed within development teams, moving some focus from implementation details toward higher-level architectural understanding and coordination. Within the scope of this thesis, however, the web implementation still requires solid knowledge of both frontend architecture and backend data structures to ensure correct development and future maintenance.

From a data perspective, the system operates on structured data that reflects the domain of the real system but does not include raw production data in its presented form. The SDV module uses data exposed through backend APIs and test or development datasets that mirror real-world structures, including organizational and technical entities. While some information is based on company-related structures, all sensitive or potentially identifiable details have been intentionally excluded or generalized in this thesis.

Finally, more modular and component-based architectures introduce a balance between improved maintainability and increased structural complexity. While separation of concerns generally improves scalability and long-term development, it also introduces more interacting parts within the system. This can increase cognitive load for developers and requires careful architectural discipline to avoid fragmented logic and inconsistent design patterns.

6.4 Future Development

The current implementation of the SDV module covers the core functionality needed for the architectural comparison, including data retrieval, filtering, and visualization. However, several planned features were not completed within the scope of this thesis. These include the optimized variables tab, the properties view, and direct database integration, along with smaller interface improvements and supporting functionality. These parts were excluded primarily due to time constraints and the focus on implementing a representative subset rather than full feature parity.

A natural next step would be to complete the remaining functionality to bring the web implementation closer to the desktop system. This would make the module more complete and better suited for real-world use beyond the scope of this comparison study.

Another improvement area lies in the data handling pipeline. While data is currently fetched through existing APIs and transformed for presentation, more advanced approaches could improve scalability and reduce complexity when dealing with larger or more complex datasets. This could include refining abstraction layers or optimizing how data is combined and processed across hooks and state management.

Further work could also focus on improving state management patterns as the system grows. Although the current separation between UI state and server state works well for the implemented scope, more complex features may require clearer conventions to avoid unnecessary coupling or fragmented logic.

Finally, a more formal evaluation of system behavior could be valuable. The current analysis is mainly qualitative, based on architectural structure and observed behavior. Introducing quantitative measurements such as rendering performance, response times, or memory usage would provide a more complete comparison between the desktop and web implementations.

In addition, the study may serve as a reference point for understanding common architectural trade-offs in similar migrations from monolithic desktop systems to component-based web architectures. However, the findings should be interpreted in the context of systems with comparable complexity and design constraints.

Overall, the current implementation is sufficient for architectural analysis, but further development would be required to make it fully production-ready and to extend its relevance beyond the scope of this thesis.

References

[1] GeeksForGeeks. *Introduction to the .NET Framework*. Available: https://www.geeksforgeeks.org/c-sharp/introduction-to-net-framework/. Accessed: 18 Mars 2026.

[2] Microsoft. *Overview of the .NET Framework*. Available: https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview. Accessed: 18 Mars 2026.

[3] Microsoft. *Windows Forms Overview* Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/. Accessed: 19 April 2026

[4] DevExpress. *WinForms Controls* Available: https://docs.devexpress.com/WindowsForms/7874/winforms-controls. Accessed: 19 April 2026

[5] Amazon Web Services. *Difference Between Monolithic and Microservices Architecture*. Available: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/. Accessed: 22 April 2026.

[6] Microsoft. *Events in Windows Forms (.NET)*. Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/forms/events?tabs=dotnet. Accessed: 22 April 2026.

[7] W3Schools. *React Intro*. Available: https://www.w3schools.com/REACT/react_intro.asp. Accessed: 21 Mars 2026.

[8] Epic react. *What is react*. Available: https://www.epicreact.dev/what-is-react. Accessed: 21 Mars 2026.

[9] React legacy. *Components and Props*. Available: https://legacy.reactjs.org/docs/components-and-props.html. Accessed: 21 Mars 2026.

[10] React. *React*. Available: https://react.dev/. Accessed: 21 Mars 2026.

[11] React legacy. *Virtual DOM and Internals*. Available: https://legacy.reactjs.org/docs/faq-internals.html. Accessed: 21 Mars 2026.

[12] React legacy. *Reconciliation*. Available: https://legacy.reactjs.org/docs/reconciliation.html. Accessed: 21 Mars 2026

[13] Loginradius. *React Hooks: A Beginners Guide*. Available: https://www.loginradius.com/blog/engineering/react-hooks-guide. Accessed: 21 Mars 2026

[14] Microsoft. *Fluent UI*. Available: https://developer.microsoft.com/en-us/fluentui. Accessed: 24 April 2026

[15] Microsoft. *Fluent UI React Storybook: Introduction*. Available: https://storybooks.fluentui.dev/react/?path=/docs/concepts-introduction–docs. Accessed: 24 April 2026

[16] React. *Composition vs Inheritance*. Available: https://legacy.reactjs.org/docs/composition-vs-inheritance.html. Accessed: 27 April 2026

[17] React. *Thinking in React*. Available: https://react.dev/learn/thinking-in-react. Accessed: 27 April 2026.

[18] Geeksforgeeks. *Component-Based Architecture - System Design*. Available: https://www.geeksforgeeks.org/system-design/component-based-architecture-system-design/. Accessed: 27 April 2026

[19] Axios. *What is Axios?* Available: https://axios-http.com/docs/intro. Accessed 19 April 2026

[20] Tanstack. *Tanstack Overview* Available: https://tanstack.com/query/v5/docs/framework/react/ Accessed 19 April 2026

[21] React. *useEffect* Available: https://react.dev/reference/react/useEffect. Accessed: 19 April 2026

[22] Mozilla. *Client-side rendering*. Available : https://developer.mozilla.org/en-US/docs/Glossary/CSR. Accessed 1 May 2026

[23] React. *createRoot*. Available: https://react.dev/reference/react-dom/client/createRoot. Accessed 1 May 2026

[24] React. *render and commit*. Available: https://react.dev/learn/render-and-commit. Accessed 1 May 2026

[25] Richards, M. and Ford, N. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[26] Refactoring Guru. *Observer Design Pattern*. Available: https://refactoring.guru/design-patterns/observer. Accessed: 24 April 2026.

[27] React. *State: A Component's Memory*. Available: https://react.dev/learn/state-a-components-memory. Accessed: 24 April 2026.

[28] Microsoft. *Application Settings Overview*. Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/advanced/application-settings-overview. Accessed: 2 May 2026.

[29] Microsoft. *How To: Write User Settings at Run Time with C#*. Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/advanced/how-to-write-user-settings-at-run-time-with-csharp. Accessed: 2 May 2026.

[30] Microsoft. *Control.Enabled Property*. Available: https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.control.enabled. Accessed: 2 May 2026.

[31] Microsoft. *Control.Visible Property*. Available: https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.control.visible. Accessed: 2 May 2026.

[32] Refactoring Guru. *Memento Design Pattern*. Available: https://refactoring.guru/design-patterns/memento. Accessed: 2 May 2026.

[33] React. *Passing Data Deeply with Context*. Available: https://react.dev/learn/passing-data-deeply-with-context. Accessed: 3 May 2026.

[34] React. *Scaling Up with Reducer and Context*. Available: https://react.dev/learn/scaling-up-with-reducer-and-context. Accessed: 3 May 2026.

[35] TanStack. *Does TanStack Query replace Redux, MobX or other client state managers?*. Available: https://tanstack.com/query/v5/docs/framework/react/guides/does-this-replace-client-state. Accessed: 3 May 2026.

[36] Dron, M. and Szandała, T. *Web Application State Management: A Review of Leading React Frameworks*. IEEE Software, vol. 43, no. 2, pp. 112–118, 2026. doi: 10.1109/MS.2025.3621269.

[37] Microsoft. What is Azure DevOps? Available: https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops. Accessed: 1 June 2026.