POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Benchmarking Modern Frontend Frameworks: A Comparative Performance Analysis

Supervisors

Prof. RICCARDO COPPOLA

Candidate

Jaouad OUCHAIB

October 2024

Summary

Recent advancements in web application development have given rise to a new breed of frontend frameworks, representing a marked departure from earlier JavaScript paradigms. These frameworks leverage declarative programming models, entrusting the framework with the task of handling UI state transitions. While this methodology streamlines development, it demands advanced rendering techniques to effectively map state modifications to UI updates. Consequently, the performance dynamics of these rendering techniques have emerged as a pivotal yet previously underexplored factor in modern web development.

This research conducts an in-depth analysis of the rendering techniques implemented by leading frameworks—namely React, Vue, Angular, and Solid—which collectively typify the current landscape of prominent web technologies. The study engages with the evolving conversation around fine-grained reactivity, particularly the utilization of signal-based systems, which contrast with conventional approaches such as virtual DOM (VDOM) reconciliation and dirty checking. By executing a series of rigorously structured benchmarks, the research quantifies and compares how rendering overhead scales with increasing application complexity. The results provide insight into the inherent performance trade-offs of each rendering strategy, demonstrating the shift towards fine-grained reactivity as a means to reduce update costs and boost rendering efficiency. In summary, this evaluation offers developers a comprehensive framework for understanding performance implications, thus facilitating more informed decisions regarding the selection of frontend technologies to meet specific scalability and efficiency criteria

Acknowledgements

I wish to express my sincere gratitude to the individuals and institutions that have supported the completion of this master's thesis.

Firstly, I extend my deepest appreciation to my academic supervisor at Politecnico di Torino, Professor RICCARDOCOPPOLA, for their invaluable supervision and insightful guidance throughout this research endeavor. Their expertise, constructive feedback, and unwavering support were instrumental in the successful completion of this work.

This thesis was developed under the academic framework of Politecnico di Torino and conducted in parallel with my professional internship at Selego. I gratefully acknowledge Selego for providing a practical context for this research within the domain of web solutions. In particular, I would like to offer special thanks to Yoan Rozak at Selego. His direct assistance with benchmark implementation, technical expertise, and insightful advice significantly enhanced the rigor and practical relevance of this study. His contributions were essential to bridging academic research with industry practices.

Furthermore, I acknowledge the contributions of my colleagues at Selego for fostering a collaborative and supportive environment during my internship, which indirectly contributed to the progress of this thesis.

The successful completion of this thesis is a direct result of the support and contributions of these individuals and institutions, to whom I am deeply indebted.

Table of Contents

List of Tables VIII

List of Figures IX

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Goal . . . 1
- 1.3 Structure of the Thesis . . . 2

2 A Historical Overview of Web Technologies 3
- 2.1 Browser Technologies . . . 3
  - 2.1.1 Browser Compatibility from Inception . . . 3
  - 2.1.2 HTML, CSS, and the DOM . . . 3
  - 2.1.3 Critical Rendering Path . . . 6
  - 2.1.4 The jQuery Revolution . . . 8
- 2.2 Design Patterns . . . 8
  - 2.2.1 Backbone.js and the Model-View-Controller (MVC) Paradigm 8
  - 2.2.2 The Paradigm Shift To Declarative UI . . . 10
  - 2.2.3 AngularJS: Declarative UI within a Model-View-Controller Context . . . 11
  - 2.2.4 The Modern MVVM: Component-Based Architectures . . . 12

3 Modern Frontend Frameworks 14
- 3.1 Framework Selection Rationale . . . 14
- 3.2 Rendering Strategies . . . 15
  - 3.2.1 Performance Considerations . . . 16
- 3.3 Frameworks Reviews . . . 18
  - 3.3.1 React . . . 18
  - 3.3.2 Solid . . . 19
  - 3.3.3 Vue . . . 22
  - 3.3.4 Angular . . . 24

3.3.5 Svelte 26

4 Performance Benchmarks 28
4.1 Goals and Methodology 28
4.2 Benchmark Descriptions 29
4.2.1 Creation of Static and Dynamic Content (Flat vs. Tree Structures) 29
4.2.2 Incremental Updates on a Flat List of Mixed Components . 31
4.2.3 Tree Update Scenarios: Root and Leaf Updates 32
4.2.4 Deeply Nested Dependency Updates: Cascading Changes . . 32
4.2.5 Interleaved Batch Operations and Concurrent Updates . . . 34

5 Performance Benchmarks: A Comparative Analysis 35
5.1 Methodology and Goals (Brief Recap) 35
5.2 Comparative Analysis of Rendering Paradigms 36
5.2.1 Creation Overhead: Static and Dynamic Content 36
5.2.2 Performance in Incremental Updates: Precision and Efficiency 38
5.2.3 Concurrency and Complex Operations 40
5.3 Intra-Framework Architectural Shifts 42
5.3.1 React 42
5.3.2 Angular 43
5.3.3 Svelte 43
5.3.4 Vue 43
5.4 Overall Conclusions 44

6 Contextualizing Benchmark Findings with Real-World Data 45
6.1 Framework Adoption and Distribution 45
6.2 Core Web Vitals Performance 47
6.2.1 Time to First Byte (TTFB) 47
6.2.2 First Contentful Paint (FCP) 48
6.2.3 Largest Contentful Paint (LCP) 49
6.2.4 Cumulative Layout Shift (CLS) 49
6.3 Correlation with Benchmark Findings 51
6.4 Limitations and Considerations 51
6.5 Implications for Framework Selection 52

7 Conclusion 53
7.1 Summary of Findings 53
7.2 Implications for Future Frontend Development 54
7.3 Compilers and AI's Potential Influence 54
7.4 Limitations of the Study 55
7.5 Recommendations for Further Research 56

A Frameworks 57

Bibliography 58

List of Tables

4.1 script execution time (ms) for rendering N static components 30
4.2 script execution time (ms) for rendering N dynamic components as a binary tree 30
4.3 Execution time (ms) updating the root component flat list N components of primarily static content (with 4x cpu slowdown) 31
4.4 Execution time (ms) updating the entire component flat list N components of primarily static content(with 2x cpu slowdown) . . 31
4.5 Execution time (ms) for root component update in a component tree of N components (with 2x CPU slowdown). 32
4.6 Execution time (ms) for leaf component update in a component tree of N components (with 2x CPU slowdown). 33
4.7 Execution time (ms) propagating an update through a dependency chain of L computations (with 2× CPU slowdown) 33
4.8 Execution time (ms) for processing interleaved batch operations and concurrent updates (2x CPU slowdown) 34

5.1 Comparison of script execution time and full render cycle time for creating N=1024 dynamic components (binary tree). 38

A.1 The list of the frameworks versions used in for the benchmarks with thier respective bundler 57

List of Figures

2.1 Example of HTML elements and structure. 4
2.2 Illustration of a DOM tree structure. 5
2.3 Example of code intercating with the CSSOM with Javascript . . . 6
2.4 Simplified illustration of the critical rendering path. 7
2.5 Placeholder for MVC Architectural Diagram illustrating the interactions between Model, View, and Controller. 9
2.6 AngularJS Architecture diagram emphasizing the data binding mechanism and its relation to the MVC pattern 12

3.1 State of JavaScript survey 2024: Adoption rates of front-end frameworks. 16
3.2 Vue 3's Runtime Reactivity and Compiler-Informed Virtual DOM (block tree) 22
3.3 Angular Component Lifecycle Diagram 25

5.1 Script execution time (ms) for creating N static components, where N is the number of components. 36
5.2 Script execution time (ms) for creating N dynamic components (binary tree). 37
5.3 Script execution time (ms) for updating the root of a static flat list. 39
5.4 Script execution time (ms) for updating all elements of a flat list. . 39
5.5 Script execution time (ms) for updating the root of a tree of N components. 40
5.6 Script execution time (ms) for deeply nested dependency updates. . 41
5.7 Script execution time (ms) under interleaved operations and concurrent updates. 41
5.8 Script execution time (ms) for updating the parent of a flat list of N=10000 elements. 42
5.9 Script execution time (ms) for swapping two rows in flat list (3x CPU slowdown) 43
5.10 Script execution time (ms) for deeply nested dependency updates. . 44

6.1 Origins over time for React, SolidJS, Svelte, Angular, and Vue. Data from [42]. 46
6.2 Percentage of sites achieving good TTFB scores. Data from [42]. . . 47
6.3 Percentage of sites achieving good FCP. Data from [42]. 48
6.4 Percentage of sites achieving good LCP scores.Data from [42]. . . . 49
6.5 Percentage of sites achieving good CLS scores. Data from [42] . . . 50

Chapter 1

Introduction

The JavaScript ecosystem is a rapidly evolving landscape, with new frameworks and libraries constantly emerging, each offering novel approaches to web development. This continuous innovation presents developers with the challenge of selecting the most suitable tools for their projects. While previous research has explored the performance of front-end frameworks, emphasizing the significance of efficient rendering strategies, the landscape has shifted.

1.1 Background

Recent developments in front-end architectures have increasingly favored the implementation of fine-grained reactivity through the use of signal primitives. This evolution, driven by objectives of boosting performance and refining developer ergonomics, is reflected in the integration of signals by leading industry players and their sustained application in frameworks such as Vue and Svelte. This transition addresses earlier findings that underscored performance constraints inherent in traditional rendering methodologies.

1.2 Goal

This thesis aims to assess the current trends in modern JavaScript front-end frameworks, particularly concerning the adoption of fine-grained reactivity and signal primitives. Through a comparative analysis, we will evaluate their performance, memory efficiency, and developer experience. This evaluation will provide developers with valuable insights and up-to-date guidance for selecting the most effective framework for their needs in 2025.

1.3 Structure of the Thesis

This thesis offers a comprehensive analysis of modern front-end frameworks, with a focus on their performance and efficiency in 2025.

- **Chapter 2** offers an in-depth overview of browser rendering mechanisms, establishing the technical foundation required for the detailed exploration of framework-specific rendering strategies in the subsequent sections .
- **Chapter 3** delves into the intricacies of popular open-source frameworks, analyzing their rendering approaches and highlighting the role of reactive programming and signal primitives.
- **Chapter 4** introduces a series of rigorous benchmarks designed to objectively evaluate the performance and memory consumption of the chosen frameworks.
- **Chapter 5** presents a meticulous analysis and comparison of the benchmark results, offering valuable insights into the strengths and weaknesses of each framework.
- **Chapter 6** contextualizes the benchmark findings using real-world data, synthesizing HTTP Archive insights and Core Web Vitals metrics to validate the controlled experiments.
- **Chapter 7** synthesizes the findings, drawing conclusions and suggesting potential directions for future research in the dynamic field of front-end development.

Chapter 2

A Historical Overview of Web Technologies

2.1 Browser Technologies

2.1.1 Browser Compatibility from Inception

Despite substantial evolution in features and complexity over decades [1], contemporary web browsers exhibit remarkable backward compatibility, capable of accurately rendering web pages designed at the very genesis of the World Wide Web, nearly thirty years ago. This persistent compatibility is rooted in the stability of the fundamental components defining a web page and the unwavering core operational principles of web browsers. Fundamentally, a web page, in both its primordial form and modern iterations, is essentially a structured document utilizing markup languages, augmented by supplementary resources. Upon receiving a URL request, a browser initiates a retrieval process followed by a series of operations to present the page visually to the user. The ensuing sections of this chapter will meticulously examine this rendering mechanism and the foundational technologies that enable it in their current state.

2.1.2 HTML, CSS, and the DOM

HyperText Markup Language (HTML), the seminal markup language for web page construction, remains the cornerstone of web content creation in its modern, evolved forms. An HTML document serves a dual purpose: to define the organizational structure and to encapsulate the information content of a web page. It achieves this through the delineation of distinct page elements and the inclusion of metadata, such as links to external resources like scripts and stylesheets. These elements

exhibit hierarchical nesting and may incorporate attributes that specify stylistic properties, classifications, or hypertextual linkages. Certain elements, notably input controls, are directly rendered and interactive for users, while others primarily provide structural or semantic context, as visually represented in Figure 2.1.

```
<!DOCTYPE html>
    <html>

        <head>
            <title> Title here </title>
        </head>

        <body>
            Web page content goes here.
        </body>

    </html>
```

Figure 2.1: Example of HTML elements and structure.

Web browsers interpret HTML by transforming it into an object-based representation known as the Document Object Model (DOM). The DOM encompasses both the tree-like hierarchical structure representing the document and the programmatic interfaces facilitating script interaction. However, common convention distinguishes between "DOM" as the data structure and "DOM APIs" for the programming interfaces, a convention this thesis will uphold. Within HTML, hierarchical document organization is achieved through nested elements denoted by tags. In the DOM, these elements are translated into nodes within a tree structure, where each node and its properties mirror an HTML element and its attributes, as illustrated in Figure 2.2.

DOM tree can be altered dynamically at runtime through the use of DOM APIs [2]. These APIs provide imperative methods to query and manipulate the tree directly, supporting operations such as selecting nodes, modifying properties, removing nodes, and inserting new ones, as illustrated in the accompanying code snippet. Additionally, because these APIs can create a variety of node types and attributes, any DOM tree produced by HTML parsing can likewise be generated programmatically

```
const sect = document.querySelector("section");
const para = document.createElement("p");
para.textContent = "We hope you enjoyed the ride.";
sect.appendChild(para);
```

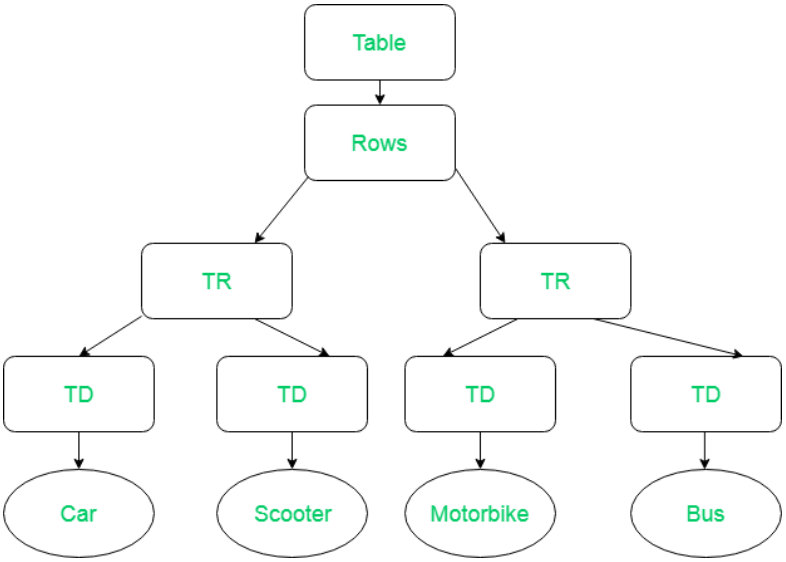

Figure 2.2: Illustration of a DOM tree structure.

The DOM specification also integrates an event handling mechanism. Specific events, such as user interactions or network responses, can trigger event dispatch. Scripts can register listener functions to react to these events, enabling dynamic and interactive behavior driven by user input. The synergy of the event system and node tree mutability empowers developers to construct sophisticated applications leveraging solely DOM APIs.

Cascading Style Sheets (CSS) is specifically engineered for defining the visual styling and presentation of documents. CSS operates on a rule-based system, applying sets of style declarations to HTML elements that match predefined selectors. Selectors can target elements based on tag names or attributes, including class and ID attributes.

Analogous to HTML, CSS is also processed into an object model, the CSS Object Model (CSSOM). While the DOM node tree represents document elements, the CSSOM is structured as a selector tree. These selectors dictate style application to DOM nodes based on their type and attributes [2]. Similar to the DOM, CSSOM provides APIs for programmatic modification of the CSSOM node tree, though these are less frequently utilized in typical web development. Common practice involves predefining style variations within CSS rules and utilizing DOM APIs to adjust DOM node attributes, thereby indirectly applying different styles by associating nodes with different CSS selectors, as illustrated in Figure 2.3.

While direct CSSOM manipulation offers theoretical performance advantages, empirical analysis of these micro-optimizations falls outside the scope of this thesis. For the current discussion, it is sufficient to recognize the prevalent practice in web

```
<style>
.initial-style {
  font-size: 16px;
}

.modified-style {
  font-weight: bold;
}
</style>
<script>
function changeStyle() {
  var element = document.getElementById("myElement");
  element.classList.remove("initial-style");
  element.classList.add("modified-style");
}
</script>
```

Figure 2.3: Example of code intercating with the CSSOM with Javascript

applications and frontend frameworks: prioritizing DOM APIs while CSS primarily serves a static styling role.

2.1.3 Critical Rendering Path

Browser vendors often refer to the "critical rendering path" as the sequential set of operations that a browser executes to convert web page resources into pixels displayed on a user's screen [3]. Although the specific labels for the various phases might differ between vendors, most major browsers follow a conceptually similar process, as illustrated in Figure 2.4 [4].

The procedure initiates with the HTML source being parsed to build the DOM tree incrementally, as the parser traverses the document. During this parsing, when the parser encounters links to external resources, it concurrently issues fetch requests while continuing to build the DOM. Once the DOM construction is complete, the associated stylesheets are parsed to generate the CSS Object Model (CSSOM)

Despite the DOM encompassing all document elements, direct rendering from the DOM is not feasible due to CSSOM rules potentially designating elements as non-visible. Consequently, upon complete construction of both the DOM and CSSOM, the browser proceeds to generate a render tree. This render tree is a refined, pruned version of the DOM, containing only elements intended for rendering, inclusive of their styling information derived from the CSSOM.

Following render tree construction, the browser calculates the dimensions and

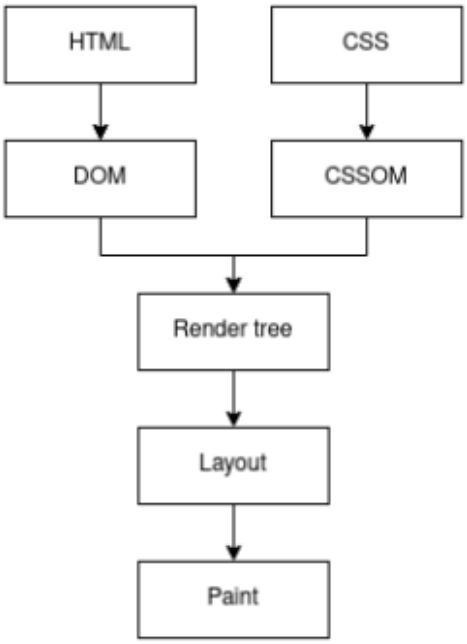

Figure 2.4: Simplified illustration of the critical rendering path.

positions of each element within it, a stage known as layout or reflow. As element visibility within the viewport cannot be determined until the complete render tree is processed, the actual painting is deferred until the entire tree layout is computed. Finally, the browser executes the painting step, rasterizing pixels onto the screen, rendering the web page visible to the user.

Scripts possess the capacity to influence and interrupt the critical rendering path. As previously discussed, scripts can utilize DOM and CSSOM APIs to dynamically modify the DOM and CSSOM node trees. Consequently, browsers, by default, halt DOM construction upon encountering a script tag. The browser then fetches, parses, and executes the script before resuming DOM construction. Asynchronous script loading, however, allows browsers to download scripts in the background without blocking DOM construction, executing them only after DOM construction is finalized.

The full critical rendering path is primarily executed on initial page load. However, various actions can trigger partial re-evaluation. DOM modifications via DOM APIs necessitate at least partial render tree reconstruction, followed by layout and paint phases. User interactions can also trigger layout and paint phases independent of render tree reconstruction. Computational cost associated with render tree construction and layout/paint phases, alongside script execution, are primary determinants of application responsiveness when the DOM is manipulated

through scripts.

2.1.4 The jQuery Revolution

Early web development was notably hindered by inconsistencies across browser implementations, particularly with respect to DOM and JavaScript support. Variations in adherence to web standards resulted in unpredictable code behavior across different platforms. jQuery emerged as a pivotal JavaScript library that directly addressed these discrepancies by abstracting browser-specific peculiarities, thereby offering a unified, cross-browser API for DOM manipulation, event handling, and AJAX interactions [5].

jQuery streamlined common JavaScript operations through a concise, expressive syntax. Tasks such as DOM element selection, which were verbose and browser-dependent in native JavaScript, were greatly simplified by leveraging a CSS-selector-based approach. Likewise, event handling was standardized across browsers, enabling more robust interactive web applications. By encapsulating and managing browser-specific quirks internally, jQuery allowed developers to concentrate on application logic rather than compatibility issues. Although jQuery maintained an imperative DOM manipulation paradigm, it substantially reduced the potential for errors associated with direct DOM access by providing a more reliable abstraction layer. However, as web applications evolved in complexity, the limitations of this imperative model in handling sophisticated UI state transitions became increasingly apparent

2.2 Design Patterns

2.2.1 Backbone.js and the Model-View-Controller (MVC) Paradigm

As the role of JavaScript in web applications expanded beyond rudimentary interactivity to encompass sophisticated application logic, the exigency for structured front-end architectures became increasingly pronounced [6]. Backbone.js emerged as a seminal framework, spearheading the adoption of the Model-View-Controller (MVC) architectural pattern in client-side development [7]. The MVC paradigm, a well-established design pattern in software engineering [8], advocates for the organization of applications through the segregation of concerns into three distinct, yet interconnected, components:

- **Model:** Encapsulates the application's data structures and business logic. The Model is responsible for data management—encompassing retrieval, persistence, and manipulation—while remaining agnostic of the user interface.

Conceptually, it represents the "what" of the application—the information domain and its governing rules.

- **View:** Constitutes the visual representation of the Model, dedicated to rendering data to the user and processing user interactions. Views focus on the "how" of data presentation. Figure 2.5
- **Controller:** Operates as an intermediary, mediating communication between the Model and the View. It manages user input originating from the View, orchestrates corresponding updates to the Model, and determines the appropriate View to render [9].

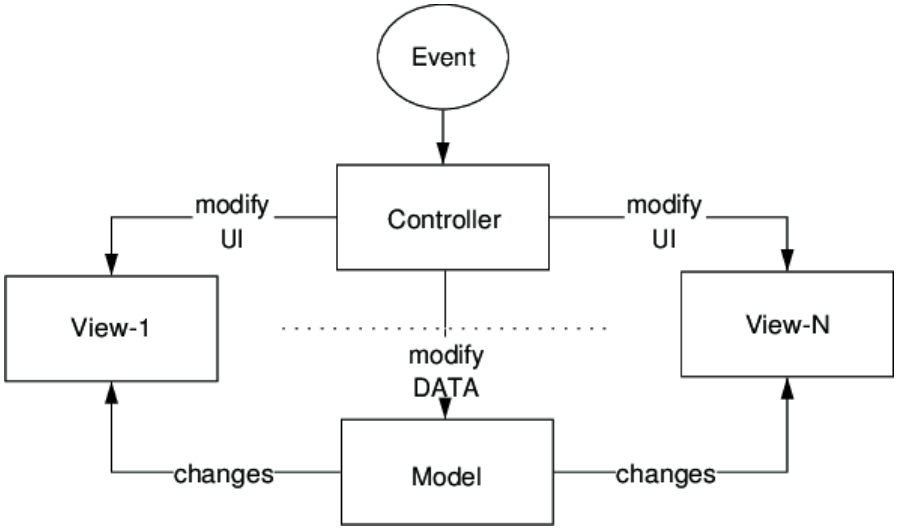

Figure 2.5: Placeholder for MVC Architectural Diagram illustrating the interactions between Model, View, and Controller.

To illustrate the MVC pattern, consider a basic online bookstore application. The **Model** would encompass entities such as book titles, pricing information, and inventory levels—along with business logic governing book searches and order processing. The **View** would manifest as the web page presented to the user, displaying book listings, search interfaces, and shopping cart functionalities. The **Controller** would manage user actions; for instance, when a user initiates a book search via the View, the Controller queries the **Model** for pertinent book data and subsequently instructs the **View** to update and render the search results.

Backbone.js offered a concrete instantiation of MVC principles within the JavaScript ecosystem [10]. It provided specific constructs for each component:

- **Models:** Backbone.js Models manage application data and are implemented as JavaScript objects equipped with event mechanisms for change notification.

Upon data modification, Models emit events, signaling associated Views to update.

- **Views:** Backbone.js Views are responsible for UI rendering, typically associated with a specific Model instance. Views subscribe to Model events and react by imperatively manipulating the DOM to reflect updated data. They also handle user-initiated events, often triggering modifications in Models or invoking Controller-like actions.

- **Routers (acting as Controllers):** Backbone.js Routers manage application navigation and URL transitions, functioning similarly to Controllers by mapping URLs to application states and corresponding Views.

Backbone.js significantly improved code organization by enforcing the principle of separation of concerns. It structured applications by encapsulating data and business logic within Models, handling UI rendering within Views, and managing application flow through Routers. This architectural approach promoted modularity, maintainability, and testability, offering a more structured alternative to the monolithic JavaScript codebases prevalent at the tim [11]. However, despite these structural advancements, Backbone.js still required developers to imperatively update the DOM within Views. For instance, upon a modification to a Model attribute, a typical View might subscribe to the corresponding change event and execute code such as:

```
this.$el.find('.book-title').text(this.model.get('title'));
```

This imperative DOM manipulation, even within the structured MVC framework, posed inherent limitations as application complexity increased. The manual management of UI state transitions, ensuring consistency between Model and View, and the risk of DOM manipulation errors persisted. Although Backbone.js laid the groundwork for data binding and event-driven architectures [12], it did not fully abstract the complexities of direct DOM manipulation—a shortcoming that eventually motivated the evolution towards more declarative front-end frameworks.

2.2.2 The Paradigm Shift To Declarative UI

Direct manipulation of the DOM via imperative APIs has been identified as a significant source of errors in web development [13]. Research by Ocariza [14] indicates that up to 80% of critical bugs in web applications stem from inaccuracies in DOM handling. This vulnerability arises from the combinatorial explosion of UI state transitions as application complexity scales [15]. In systems with N distinct

UI states and k possible transitions from each state, the total number of valid transitions rapidly escalates to

$$kN(N-1)$$

. Imperative DOM manipulation necessitates explicit definitions for each transition, amplifying the potential for errors such as operating on non-existent DOM nodes.

Modern web frameworks address these challenges by adopting a declarative programming paradigm for UI development. Rather than detailing how to update the DOM, developers specify the desired UI state, and the framework automatically manages the underlying DOM operations required to achieve that state [16]. This abstraction not only reduces the risk of errors but also streamlines development by allowing developers to focus on application logic and UI design rather than low-level DOM manipulation. However, this approach does introduce framework overhead in computing the necessary DOM operations for each state transition [17]. This overhead is closely related to solving the tree edit distance problem [18], where the framework computes the minimal set of operations (add, delete, update) to efficiently transform the current DOM tree into the desired one.

2.2.3 AngularJS: Declarative UI within a Model-View-Controller Context

AngularJS emerged as a pioneering framework that embraced the declarative UI paradigm while conceptually aligning with the MVC pattern [19]. AngularJS directly addressed the limitations of imperative view updates found in earlier MVC implementations like Backbone.js by introducing declarative data binding mechanisms. As illustrated in Figure 2.6, AngularJS allowed developers to define HTML templates where UI elements were directly bound to Model data.

In AngularJS, changes in the Model were automatically reflected in the View through two-way data binding [20], and user interactions within the View could automatically trigger updates to the Model. This approach greatly reduced the need for manual DOM manipulation and significantly mitigated errors associated with imperative updates. Although AngularJS retained core MVC principles, its introduction of the "Scope" object blurred the lines between traditional MVC and MVVM, with some interpretations viewing AngularJS as an early form of MVVM due to its declarative binding features [21].

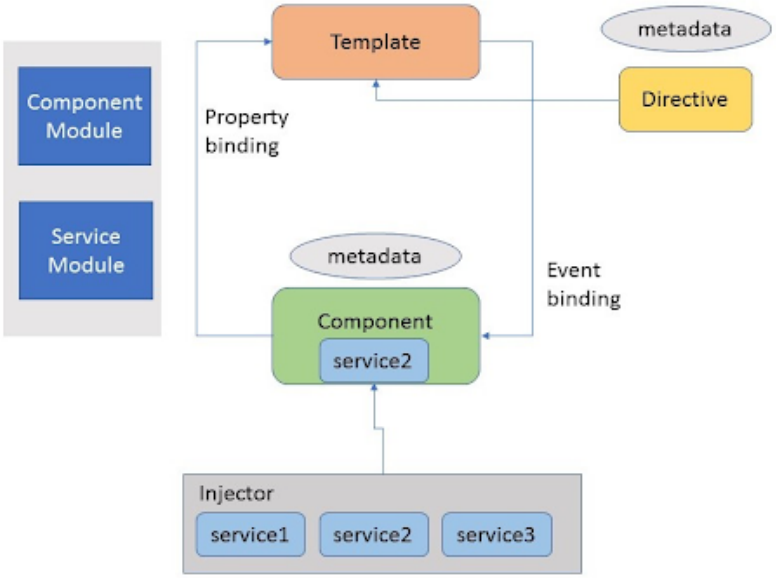

Figure 2.6: AngularJS Architecture diagram emphasizing the data binding mechanism and its relation to the MVC pattern

2.2.4 The Modern MVVM: Component-Based Architectures

While AngularJS marked a significant move towards declarative UI within an MVC framework, the next major paradigm shift occurred with the rise of component-based architectures, as epitomized by React [22]. Instead of segregating concerns strictly into Model, View, and Controller, modern frameworks like React organize applications into independent, reusable components [23]. Each component encapsulates its own:

- **View Logic (Rendering):** How the component visually represents itself.
- **State Management:** The internal data and UI state specific to that component.
- **Behavior and Interactions:** How the component responds to user events and other inputs.

In this context, components serve a similar role to ViewModels in the MVVM pattern, but the focus shifts from a global separation of concerns to dividing the application by distinct business logic domains (e.g., shopping cart, dashboard) [24].

For instance, in an e-commerce application, one might implement components such as:

- `ProductList`: Displays a list of products.
- `ShoppingCart`: Manages the user's shopping cart.
- `ProductDetails`: Holds detailed information about a single product.
- `CheckoutForm`: Handles the checkout process.

Each component encapsulates its own rendering logic, data requirements, and interactions, thereby promoting modularity, reusability, and ease of testing [25]. This component-based paradigm represents the modern evolution of MVVM, where each component effectively functions as a self-contained ViewModel, streamlining state management and UI updates through a declarative approach [26].

The next chapter will delve deeper into React and other component-based frameworks, further exploring how this model diverges from traditional MVC and the specific benefits it offers for modern web application development.

Chapter 3

Modern Frontend Frameworks

This chapter presents a comparative analysis of Angular, React, Vue, Svelte, and Solid, focusing on their rendering strategies and performance characteristics. We delve into the evolution of rendering strategies employed by these frameworks.

3.1 Framework Selection Rationale

Building on the MVVM paradigm discussed in Chapter 2, it is evident that modern frontend frameworks uniformly incorporate declarative data binding and component-based architectures. Figure 3.1 adapted from the State of JavaScript survey 2024 underscores that the selected frameworks are among the most prevalently adopted in the industry. This empirical evidence reinforces our framework selection, which is strategically grounded in both innovation and widespread usage.

The frameworks included in this study were chosen based on several key criteria:

- **Empirical Relevance:** As shown in Figure 3.1, the selected frameworks (React, Angular, Vue, Svelte, and Solid) dominate the front-end ecosystem, whereas alternatives such as Preact—being mere variations of React with nuanced paradigm shifts—were excluded to maintain analytical clarity.
- **Innovative Rendering Mechanisms:** The chosen frameworks encompass a spectrum of rendering strategies—from virtual DOM diffing (React, Angular, Vue) to compile-time optimizations (Svelte) and fine-grained reactivity (Solid)—which are at the forefront of addressing performance and memory efficiency challenges.
- **Industrial Adoption and Maturity:** Each framework demonstrates robust

industrial usage and maturity, making them ideal candidates for a comparative analysis of their technical merits.

- **Scope Limitation:** Frameworks based on alternative paradigms, such as Blazor and other WebAssembly (Wasm)-based solutions, are deliberately excluded from this study due to their fundamentally different operational models and the resultant divergence in evaluation metrics.

While a detailed technical exposition of each framework is provided in subsequent sections of this chapter, the following high-level observations motivated their inclusion:

- **React** exemplifies a robust component-driven model with a virtual DOM that emphasizes unidirectional data flow, contributing to predictable UI behavior.
- **Angular** integrates comprehensive tooling and bidirectional data binding, which, despite inherent overhead, supports large-scale enterprise applications.
- **Vue** offers a balanced and flexible approach to reactivity and component-based design, facilitating incremental adoption across projects of varying sizes.
- **Svelte** and **Solid** introduce cutting-edge paradigms—via compile-time optimizations and fine-grained reactivity, respectively—thereby challenging traditional virtual DOM methodologies.

The forthcoming sections will provide an in-depth, technical exposition of each framework's architecture, performance implications, and memory efficiency, thereby extending the foundational concepts introduced here.

3.2 Rendering Strategies

While all frameworks aim to synchronize the DOM with the component tree, they employ different rendering strategies:

Virtual DOM: These frameworks explicitly solve the tree edit distance problem [18] by comparing a virtual representation of the DOM (vDOM) with the actual DOM. This involves generating a new vDOM tree based on the current application state, comparing it with the previous vDOM tree, and applying the minimal set of changes to the real DOM.

Implicit Tree Diffing : These frameworks implicitly solve the tree edit distance problem through dirty checking of data bindings. Each component tracks its data bindings and updates the corresponding DOM elements when changes are detected.

Fine-grained Reactivity: These frameworks bypass explicit tree diffing by establishing a direct link between data changes and DOM updates. They utilize

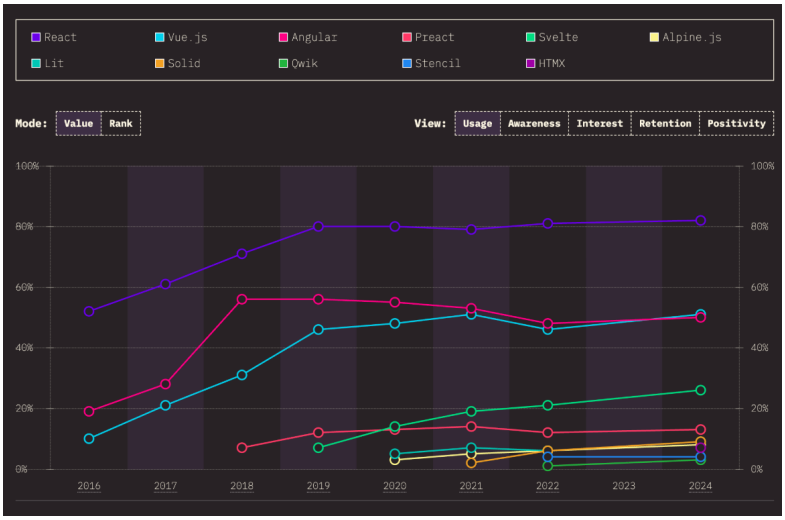

Figure 3.1: State of JavaScript survey 2024: Adoption rates of front-end frameworks.

dependency tracking mechanisms to pinpoint and update only the affected DOM elements, resulting in highly efficient rendering.

3.2.1 Performance Considerations

The frameworks examined in this study all utilize a loop mechanism to traverse the component tree, resulting in a baseline $O(n)$ time complexity for each iteration, where n is the number of nodes in the tree. However, significant performance variations emerge due to differences in input sizes and the fixed costs inherent in each framework's implementation.

Fixed Costs

Quantifying the fixed costs associated with each framework is challenging. While all frameworks manage component lifecycles and data binding, the specific operations performed per component and binding are heavily influenced by internal implementation details.

An example of a fixed cost is in dirty checking. This involves storing a copy of a value and later comparing it against the current value to detect changes. Dirty checking is frequently employed for data bindings. The framework stores a copy of a binding's value when it's initially rendered to the DOM and subsequently compares this copy to the current value to determine if an update is necessary. Dirty checking objects can be complex and computationally expensive, especially

when dealing with nested objects and the need to account for potential mutations of object properties.

Another fixed cost factor is the choice of templating engine. This selection involves significant trade-offs. JSX, a domain-specific language (DSL) that allows for XML-like syntax within JavaScript[27], provides runtime interpolation and full JavaScript expressiveness. However, this dynamic nature hinders ahead-of-time (AOT) optimizations[28], potentially impacting performance. Runtime scheduling can improve perceived performance but introduces a larger runtime code footprint and increased complexity.

Conversely, static templates offer the compiler more opportunities for optimization, leading to better raw performance and reduced bundle sizes. However, they come at the cost of a more constrained syntax and impose stricter requirements on the build process.

Input Size

Input size, the second major factor influencing performance, encompasses the number of components, static elements, and data bindings per component. During the initial creation of the component tree, all components, elements, and bindings are processed, resulting in equivalent input sizes across frameworks. However, when updating existing components, input sizes can diverge significantly due to variations in rendering strategies and the amount of the component tree they re-render. The smaller the number of components and elements a framework checks for updates during a change cycle, the better its performance will generally be. Furthermore, the use of a virtual DOM introduces additional overhead, as it necessitates the explicit calculation of DOM changes. This often requires two loops: one to construct a new virtual DOM tree and another to compare it against the previous tree, generating the necessary DOM API calls for updates.

Frameworks like Angular and React rely on a pull-based reactivity system[29]. This means they require an explicit signal to initiate an update cycle where changes are detected and propagated to the DOM (e.g., React's ‘setState‘ function). While frameworks can intelligently trigger these signals based on asynchronous events (e.g., Zone.js in Angular), this approach can lead to unnecessary re-computations because the framework lacks visibility into which parts of the component tree are truly static versus dynamic, thus resulting in larger input sizes during each update cycle. Mitigating this often requires manual optimization hints from the developer (e.g., ‘memo‘ in React or ‘OnPush‘ in Angular).

In contrast, other frameworks implement a push-based system [30] operating at a finer granularity – individual data bindings. By intercepting property getters and setters, these frameworks track dependencies and establish reactive units. This allows the system to more precisely identify changes, ideally leading to smaller

input sizes during updates as only affected sections of the component tree are processed. This eliminates much of the need for broad component re-computation. However, this approach introduces the overhead of maintaining a dependency tracking graph. Each tracked reactive property is typically stored within a closure along with a collection of its dependents. This can lead to increased memory allocation, particularly with large datasets. Using immutable objects can mitigate this issue, as the system can then safely assume that specific object graphs do not require observation.

3.3 Frameworks Reviews

3.3.1 React

React[22], developed by Facebook and open-sourced in 2013, changed forever the frontend development by explicitly addressing the tree edit distance problem [31]. At its core lies the Virtual DOM (vDOM), an in-memory representation of the actual DOM. React applications are structured as trees of components, each defining a `render` function that produces a vDOM node, describing the desired UI state. For example, a simple component's `render` function might look like this:

```
function MyComponent(props) {
  return (
    <div>
      <h1>Hello, {props.name}!</h1>
    </div>
  );
}
```

The framework's rendering loop involves traversing this component tree, invoking `render` functions, and then performing *reconciliation*[31]. This process compares the newly generated vDOM with the previous one to determine the minimal set of changes required to update the real DOM, effectively solving the tree edit distance problem by efficiently updating the user interface based on state changes. Initially, React adopted a pull-based change detection, requiring manual triggers from developers to initiate these render loops.

React's architecture is deeply rooted in the principle that everything is data, promoting a unidirectional data flow and a declarative programming style. This is evident in how components manage state and how UI updates are driven by data changes. For instance, state updates in React components trigger re-renders:

```
function Counter() {
  const [count, setCount] = React.useState(0);
```

```
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

While conceptually simple and widely adopted, this initial approach presented performance challenges. React treated all content uniformly, necessitating processing of both static and dynamic elements during each render cycle. Furthermore, by default, component updates triggered re-rendering of entire subtrees, potentially leading to inefficiencies even when child components remained unchanged. Although developers could manually optimize performance through techniques like component skipping and render function memoization, the framework itself lacked inherent optimizations to differentiate between static and dynamic content automatically. This mental model of "everything re-renders" simplified development but highlighted areas for potential performance improvement, especially in complex applications.

The React team introduced recently a Compiler[32] marking a significant evolution in the framework, driven by the need to overcome the inherent performance limitations of its original rendering strategy. While the Virtual DOM and reconciliation offered a powerful abstraction, the overhead of runtime diffing and processing, even for static content, became increasingly apparent. The React Compiler addresses these shortcomings by automatically optimizing components at build time. By leveraging memoization and static analysis, the compiler aims to minimize unnecessary re-renders, enabling React to achieve finer-grained updates and significantly improve runtime performance without burdening developers with manual optimization in many common scenarios. This shift towards compilation represents an effort to retain React's core philosophical strengths – developer experience and a predictable mental model – while achieving greater efficiency and performance competitiveness in modern frontend applications, moving beyond the initial reliance on runtime tree diffing as the primary optimization strategy.

3.3.2 Solid

SolidJS [33] stands out as a framework that has significantly influenced modern frontend trends, notably by popularizing signals as its core reactivity primitive and showcasing exceptional performance in micro-benchmarks. SolidJS champions a fine-grained reactivity system built around signals, offering a compelling alternative

to Virtual DOM-based frameworks. Signals[34], in SolidJS, are reactive primitives that encapsulate values and automatically track dependencies. This design choice allows SolidJS to achieve highly efficient updates by directly targeting only the components and DOM elements affected by data changes. SolidJS departs from the Virtual DOM approach entirely, opting for a compilation strategy that generates imperative DOM update code, inspired by techniques like "Imperative Codegen" observed in projects like Vue Vapor, as illustrated in the provided code snippet. This direct manipulation minimizes runtime overhead and contributes to SolidJS's performance profile.

SolidJS uses a reactivity model often described as a "hybrid push-pull" system, centered around signals. At its heart, a SolidJS signal can be conceptually understood using the simplified implementation provided:

```
let \_count = 0;
const count = () => {
  track();
  return \_count;
}
const setCount = (val) => {
  \_count = val;
  trigger();
}
// Usage with Solid's API:
// const [count, setCount] = createSignal(0);
```

In this conceptual model, ‘track()‘ registers the current reactive context (e.g., a component's render function or an effect) as a dependency of the signal. ‘trigger()‘ is then responsible for notifying all tracked dependencies when the signal's value is updated, initiating targeted updates. SolidJS efficiently manages these dependencies, creating a fine-grained reactivity graph. When a signal's value changes via ‘setCount‘, only the components and effects that explicitly access the signal through ‘count()‘ are re-executed, leading to highly localized and performant updates. This push-based notification is coupled with a pull mechanism within components and effects: when re-executing, they "pull" the latest signal values using accessors like ‘count()‘, ensuring data consistency.

In contrast, React's core insight is that initialization and updates should be treated as the same problem. The user doesn't care when the UI is rendered, only that it accurately reflects the current state. React achieves this by making *all* rendering logic reactive by default. Developers can write straightforward, imperative code, using standard JavaScript control flow (if statements, loops, etc.), without having to manually manage dependencies or restructure their code around individual values.

The following React code for example:

```
// Working version in React
function VideoList({ videos, emptyHeading }) {
    const count = videos.length;
    let heading = emptyHeading;
    let somethingElse = 42;
    if (count > 0) {
        const noun = count > 1 ? 'Videos' : 'Video';
        heading = count + ' ' + noun;
        somethingElse = someOtherStuff();
    }
    return (
    <>
        <h1>{heading}</h1>
I       <h2>{somethingElse}</h2>
    </>
    );
}
```

This code naturally expresses the rendering logic. In a purely fine-grained system (without compiler optimizations), achieving the same reactivity often requires restructuring the code around individual values, potentially leading to less intuitive code organization:

```
// Working version in Solid
function VideoList(props) {
    const count = () => props.videos.length;
    const heading = () => {
    if (count() > 0) { // can't add the logic here :(
        const noun = count() > 1 ? "Videos" : "Video";
        return count() + " " + noun;
    } else {
        return emptyHeading;
    }
}
const somethingElse = () => {
    if (count() > 0) { // let's put this here i guess
        return someOtherStuff();
    } else {
        return 42;
    }
});
return (
    <>
        <h1>{heading()}</h1>
        <h2>{somethingElse()}</h2>
```

```
    </>
);
}
```

While both versions achieve the desired result, the Solid example requires a different mental model, where reactivity is explicitly managed through functions (or signals). The React code, while potentially less performant without compiler optimizations, arguably offers a more natural and declarative way to express the rendering logic. React's approach is to "make rendering logic reactive by default", whereas in the pre-compiled Solid code, the programmer must *explicitly* define what is reactive.

3.3.3 Vue

Vue, positioned as a progressive framework, adopts a dual rendering strategy leveraging both a Virtual DOM (vDOM) and template compilation to optimize performance. Similar to React, Vue [35] utilizes a vDOM and reconciliation to address the tree edit distance problem. Vue components, syntactically defined through templates or render functions, ultimately produce vDOM nodes. These templates, reminiscent of Angular's directive-based syntax, are in fact "syntactic sugar" that are parsed and compiled into optimized render functions. During updates, Vue performs reconciliation, comparing the current and previous vDOM to calculate minimal DOM changes. Vue's early versions, like React, employed a vDOM reconciliation approach conceptually akin to React, as illustrated by a basic component structure:

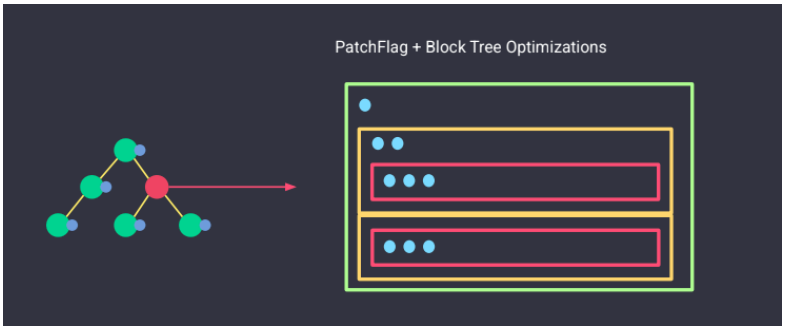

Figure 3.2: Vue 3's Runtime Reactivity and Compiler-Informed Virtual DOM (block tree)

Vue distinguishes itself through its incorporated reactivity system, a push-based mechanism that automatically tracks dependencies and optimizes updates. Unlike React's initial pull-based approach, Vue implements a reactive programming model using proxies. When data becomes reactive in Vue, it's converted into proxy objects. Accessing properties of these proxies within reactive contexts, such as

render functions or computed properties, establishes dependencies. Vue's reactivity system then automatically tracks these dependencies. When a reactive data value changes, only the components or parts of the template that depend on that specific data are re-evaluated and updated. This fine-grained reactivity, as demonstrated in Vue's reactivity setup, ensures efficient updates by only processing components with explicit dependency changes, effectively mimicking a component tree with `shouldComponentUpdate` implemented everywhere:

```
import { ref, effect } from 'vue'

const count = ref(0)

effect(() => {
  console.log('Count is:', count.value)
})

count.value++ // Triggers the effect to re-run
```

This push-based reactivity inherently reduces the input size for update loops, approaching optimal levels by checking only dirty components and their data bindings.

Despite Vue's optimized vDOM implementation and reactivity system, the overhead associated with maintaining a vDOM remained a point for further optimization. To address this, Vue introduced Vue Vapor, an experimental rendering mode employing "Imperative Codegen," as seen in Solid and other frameworks. Vue Vapor bypasses the vDOM entirely for optimized templates. Instead of reconciliation, Vapor compiles templates directly into highly optimized imperative JavaScript code that manipulates the DOM directly. This approach, exemplified by code generation techniques, eliminates the overhead of vDOM diffing, leading to significant performance gains, especially in scenarios with frequent updates. While Vue retains the vDOM for compatibility and flexibility, Vapor represents a move towards even greater raw performance, prioritizing direct DOM manipulation for templates where the vDOM abstraction might introduce unnecessary overhead.

```
import { ref, effect } from 'vue'
import { template, on, setText } from 'vue/vapor'
const t0 = template(`<div><button>`)
export default () => {
    const count = ref(0)
    let div = t0()
    let button = div.firstChild
    let _button_text
    effect(() => {
        setText(button, _button_text, _button_text = count.value)
    })
```

```
    on(button, 'click', () => count.value++)
    return div
}
```

3.3.4 Angular

Angular [36], the successor to AngularJS, represents a complete reimagining as a compiler-based framework. Built with TypeScript, Angular applications are compiled into JavaScript, augmented by a runtime environment. A key architectural decision in Angular was to address the cyclical dependency issues inherent in AngularJS by implementing one-way data binding and enforcing component-local state by default, unless explicitly shared. This design choice transforms the update mechanism from a potentially cyclic graph to a tree, ensuring a deterministic $O(n)$ time complexity for each update loop, where n is the number of nodes in the component tree. An Angular component is composed of a TypeScript class and an HTML template (extended with directives), which the Angular compiler transforms into a runtime component definition. At the heart of this definition lies the template function, responsible for rendering and updates. This function, generated by the Angular compiler, contains distinct branches for initial rendering and subsequent updates, optimizing for performance.

Angular's original rendering strategy, prior to Signals, relied on a pull-based change detection system. A render loop in Angular traverses the entire component tree, calling each component's template function. For initial rendering, components are created and with thier bindings. For updates, it employed dirty checking: for each binding within a component, Angular compared the current value against a previously stored value, updating the DOM only when changes were detected.The Angular compiler optimizes the template function such that the update branch only processes data bindings, effectively bypassing static content and reducing the overhead per component. To automatically trigger these render loops, Angular relies on Zones.js, an execution context that monkey patches asynchronous browser events. While Zones.js automatically initiates change detection upon events like user interactions or HTTP responses, it lacks fine-grained knowledge of which components truly require updates. This necessitates checking the entire component tree during each update cycle, even if only a small portion of the application state has changed, as depicted in Angular's component lifecycle: This pull-based, full-tree change detection, while deterministic, could become inefficient in larger, more complex applications.

The introduction of Signals in Angular represents a significant shift towards a more fine-grained and performant reactivity model [37]. Signals introduce a publish-subscribe pattern, allowing components to subscribe to specific data changes directly. This mechanism bypasses the need for full component tree traversal for every update.

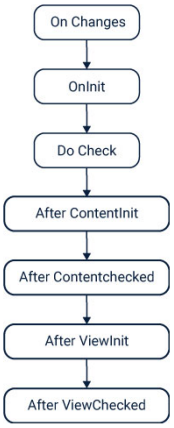

Figure 3.3: Angular Component Lifecycle Diagram

With Signals, components are updated directly when their subscribed signal values change, eliminating the overhead of checking unchanged parts of the component tree. This addresses key limitations of the previous automatic change detection, including the performance cost of unnecessary full-tree checks and issues like the "ExpressionChangedAfterItHasBeenCheckedError". Using Signals, as illustrated in the following example, enables developers to guide Angular to update only the necessary parts of the UI, leading to potential performance gains, particularly in large applications:

```
import { Component, signal } from '@angular/core';

@Component({ ... })
export class MySignalsComponent {
  total = signal(1);

  multiplyByTwo() {
    this.total.update((val: number) => val *= 2);
  }

  // ...
}
```

Signals, while adding a new layer of complexity and requiring a shift in mental model for some Angular developers, offer a path to significantly enhance application performance by enabling more targeted and efficient change detection, moving beyond the limitations of automatic, full-tree traversal and paving the way for finer-grained updates reminiscent of push-based reactivity systems seen in other frameworks.

3.3.5 Svelte

Svelte [38] presents a unique approach as a compiler-first framework, fundamentally shaping both its reactivity model and overall architecture. Unlike runtime-heavy frameworks, Svelte components are transformed during compilation into highly optimized JavaScript code that directly manipulates the DOM, eschewing the Virtual DOM entirely. Svelte's reactivity is also compiler-driven: reactive declarations, marked with `$` syntax, are analyzed at compile time to build dependency graphs. The compiler then injects fine-grained update logic, ensuring that only the parts of the DOM affected by state changes are updated, resulting in efficient, surgically precise DOM manipulations. This philosophy of compile-time optimization underpins Svelte's core identity, aiming for lean, performant applications with minimal runtime overhead. A Svelte component showcasing this reactive syntax appears as:

```
<script>
  let count = 0;
  $: doubled = count * 2; // Reactive declaration - compiler tracks dependencies

  function increment() {
    count += 1;
  }
</script>

<button on:click={increment}>
  Count: {count}, Doubled: {doubled}
</button>
```

Svelte 5 marked a significant evolution with the introduction of "runes," a more explicit and fine-grained reactivity system [39]. While previous versions relied on compiler magic to infer reactivity, runes provide developers with more direct control and predictability over reactive behavior. Runes like `$state`, `$derived`, and `$effect` offer explicit APIs to declare state, derived values, and side effects, respectively. This shift addresses some challenges inherent in compiler-centric frameworks. In large codebases, implicit compiler behavior could sometimes become opaque, making it harder to trace reactivity and debug complex interactions. Furthermore, relying heavily on compiler inference might lead to subtle API clashes or unexpected behavior in increasingly intricate projects as teams grew and project scale increased. Runes mitigate these issues by making reactivity more explicit and developer-controlled, reducing the "magic" and improving the understandability and maintainability of Svelte applications, particularly in large-scale projects, giving developers finer control as highlighted with `$effect.tracking`.

The move to runes in Svelte 5 is not a departure from its compiler-first philosophy but rather a refinement. It acknowledges the trade-offs between compiler-driven implicit reactivity and developer control. By providing explicit reactivity APIs,

Svelte empowers developers with better tools to manage reactivity in complex applications, potentially improving predictability and debuggability without sacrificing the performance benefits of compile-time optimizations. While Svelte still relies heavily on its compiler for generating highly efficient code, runes represent a step towards a more balanced approach, offering a more transparent and manageable reactivity model, especially crucial for larger teams and codebases where explicitness and predictability become paramount concerns for long-term maintainability and collaborative development. This evolution demonstrates Svelte's ongoing effort to refine its approach, balancing performance with developer experience and addressing the practical challenges of building large, maintainable frontend applications with a compiler-centric framework.

Chapter 4

Performance Benchmarks

This chapter details a suite of CPU-centric benchmarks designed to empirically evaluate performance differences between virtual DOM-based rendering, and various fine-grained reactive systems, including different versions of frameworks that have evolved their reactivity models, and the effect of compiler optimizations. The focus is on script execution time, isolating the computational overhead of the frameworks themselves from browser-specific rendering phases (layout, paint). This provides a precise measure of each paradigm's inherent update mechanism efficiency.

4.1 Goals and Methodology

As discussed in previous chapter, script execution costs for different rendering strategies are hypothesized to diverge significantly based on the rendering loop: creation of new components versus updating existing components. Updates to components with substantial subtrees, or those with primarily static content, are expected to show pronounced differences.

The primary objective is to rigorously validate these hypothesized differences, providing quantitative evidence. These benchmarks quantify the relative magnitudes of performance variations, focusing on the evolution within frameworks and the impact of compiler enhancements.

Due to the simplified nature of benchmarked components, results are comparative approximations, highlighting relative efficiencies of underlying update mechanisms, not absolute measures of real-world application performance.

Traditional web front-end framework assessments often include the entire render cycle [40]. This conflates framework-specific computations with browser overhead. While layout and paint are relevant to overall performance, they are orthogonal to the framework's core computational work. This study meticulously measures both the complete render cycle duration and, crucially, the isolated script execution

time.

The absence of a standardized API for precise script execution time measurement necessitates a platform-specific approach. These benchmarks leverage the Chrome Devtools Protocol (CDP) in Chrome-based browsers (specifically, Chrome version 131)[41]. CDP enables programmatic control and tracing of script execution. Tracing, based on CPU polling at 200-microsecond intervals, yields millisecond-level precision. Each benchmark is executed 10 times for each framework (and version, where applicable), preceded by 5 warm-up runs. The mean of the 10 measured values is reported, along with the standard deviation to indicate measurement consistency.

Benchmarks were performed on a Mac Book Air 13 with M3 chip 8-core CPU, 16 GB of RAM. Interaction with CDP was achieved using Puppeteer.

For each scenario, isomorphic applications are implemented in each framework/version. A user action triggers a render cycle, during which script execution time is measured. Scenarios probe distinct aspects of rendering performance by varying component tree structure and the nature of executed actions.

Benchmarks were performed for all frameworks discussed, with a focus on comparing versions where significant reactivity model changes occurred, or where compiler optimizations were introduced. A representative subset of results across a range of input sizes is presented. Comprehensive results, including full render cycle durations and standard deviations, are in Appendix B.

4.2 Benchmark Descriptions

The following benchmarks elucidate performance trade-offs between re-rendering entire component trees and employing fine-grained reactive updates, with a particular emphasis on how these trade-offs have changed with framework evolution and compiler optimizations.

4.2.1 Creation of Static and Dynamic Content (Flat vs. Tree Structures)

This benchmark set quantifies cost differences between creating static elements and dynamic components, and assesses the influence of component tree structure (flat vs. hierarchical) on creation performance.

- 1. Static Components: Time to render N static elements. A single component directly renders these N elements, establishing a baseline rendering cost for static content.

- 2. Component Overhead (Flat): A single parent component with N child components. Each child outputs a single static element. This isolates the computational cost of component construction.

- 3. Component Overhead (Tree): A binary tree structure (to assess the impact of a large number of children), where each non-leaf component has two children, totaling N components. Each component renders a single static element. Each component also incorporates a minimal dynamic binding (a boolean flag toggled by a button, conditionally rendering children) to observe how this cost scales in a hierarchical structure.

Table 4.1: script execution time (ms) for rendering N static components

Framework	N=1000	N=5000	N=10000	N=25000	N=50000
Angular l	15	42	79	234	682
Angular (with signals)	12	39	65	194	423
React	8	109	224	924	3536
React (with compiler)	9	126	249	966	3659
Vue	7	26	62	173	245
Vue vapor	7	31	59	93	229
Svelte 4	8	28	69	165	458
Svelte 5	3	13	25	63	104

Table 4.2: script execution time (ms) for rendering N dynamic components as a binary tree

Framework	N=512	N=1024	N=4096	N=8192	N=16384
Angular	75	120	216	469	774
Angular(with Signals)	73	130	202	479	725
React	32	56	135	379	709
React (with Compiler)	32	55	137	394	733
Vue 3	48	73	206	465	757
Vue vapor	44	75	202	468	798
Svelte v4	19	71	194	432	623
Svelte v5	17	67	102	284	509
Solid	19	61	108	279	499

4.2.2 Incremental Updates on a Flat List of Mixed Components

This focuses on efficiently updating components in a flat list. Components are mostly static, except for a single dynamic value (a counter) within each.

The primary measurement is the time to update only this dynamic value. This isolates the cost of targeted updates, contrasting fine-grained reactivity (only the changed value is updated) with potential full component recomputes.

"Add one," "delete all," and "swap two items" operations are included, measuring responsiveness to incremental and mass modifications, and common list manipulation. This scenario is representative of dynamically updated lists/tables.

Table 4.3: Execution time (ms) updating the root component flat list N components of primarily static content (with 4x cpu slowdown)

Framework	N=500	N=1000	N=5000	N=10000	N=25000	N=50000
Angular	21	52	91	145	216	432
Angular(with Signals)	12	29	68	101	184	341
React	162	246	403	623	894	1547
React (with Compiler)	105	124	230	321	486	613
Vue	2	3	5	8	12	56
Vue vapor	17	29	37	67	75	126
Svelte v4	3	7	10	14	16	42
Svelte v5	1	1	3	5	8	14
Solid	<1	2	3	4	6	12

Table 4.4: Execution time (ms) updating the entire component flat list N components of primarily static content(with 2x cpu slowdown)

Framework	N=500	N=1000	N=5000	N=10000	N=25000	N=50000
Angular	16	34	52	56	81	476
Angular(with Signals)	11	26	36	39	61	296
React	84	123	204	456	544	1681
React (with Compiler)	93	112	223	376	523	1733
Vue	62	83	146	182	305	637
Vue vapor	44	75	102	209	368	598
Svelte v4	6	10	20	40	108	164
Svelte v5	5	67	102	213	284	509
Solid	5	61	108	196	279	499

4.2.3 Tree Update Scenarios: Root and Leaf Updates

This investigates update propagation in a hierarchical binary tree (representing, e.g., nested menus). Most nodes are static; a few have dynamic values. This analyzes update behavior at different tree levels.

Time is measured for updates at:

- Root Node: Potentially triggers a full tree update (worst-case).
- Leaf Node: Isolated, localized update (ideal for fine-grained reactivity).

Comparing these reveals whether updates remain localized (efficient fine-grained reactivity) or propagate unnecessarily. Only one node in each branch contains a dynamic value.

Table 4.5: Execution time (ms) for root component update in a component tree of N components (with 2x CPU slowdown).

Framework	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	23	32	65	69	88	212
Angular (with Signals)	14	21	52	59	81	201
React	46	84	184	296	422	758
React (with Compiler)	49	92	195	314	462	778
Vue	1	1	1	2	2 2	
Vue vapor	1	1	1	1	1	1
Svelte 4	1	1	1	1	2	2
Svelte 5	1	1	1	1	1	1
Solid	1	1	1	1	1	1

4.2.4 Deeply Nested Dependency Updates: Cascading Changes

This benchmark evaluates the performance of propagating an update through a long chain of dependent computations. In this scenario, each computation in the chain is linked to the next so that an update at the root triggers a cascade of updates through every subsequent node. The chain lengths examined are 10, 50, 100, 200, 500, and 1000 nodes.

The primary goal is to assess the cost of explicitly traversing a dependency chain versus the cost of rebuilding a subtree without explicitly iterating over each node. In our tests, both fine-grained reactive systems and full re-render models (e.g., React) employ batching strategies to coalesce updates. As a result, the measured execution times show only marginal differences between the two approaches. Even though

Table 4.6: Execution time (ms) for leaf component update in a component tree of N components (with 2x CPU slowdown).

Framework	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	26	28	66	66	88	208
Angular (with Signals)	25	27	65	62	71	201
React	1	2	7	7	11	9
React (with Compiler)	1	3	9	8	12	10
Vue	1	1	1	1	2	2
Vue vapor	1	1	1	1	1	2
Svelte v4	1	1	1	1	1	2
Svelte v5	1	1	1	1	1	1
Solid	1	1	1	1	1	1

one might expect that traversing a long chain would result in a deep call stack and higher latency in fine-grained systems, the overhead is effectively amortized by batching—making the cost of processing a long dependency chain similar to the cost of rebuilding a subtree in a batched full re-render.

The results below (measured under a 2× CPU slowdown) indicate that across all frameworks, the execution time scales modestly with chain length.

Table 4.7: Execution time (ms) propagating an update through a dependency chain of L computations (with 2× CPU slowdown)

Framework	L=10	L=50	L=100	L=200	L=500	L=1000
Angular	7	10	15	36	49	69
Angular (with Signals)	7	10	14	31	42	61
React	4	6	9	13	29	52
React (with Compiler)	4	6	9	13	28	51
Vue	5	5	11	22	31	48
Vue vapor	4	6	9	17	27	39
Svelte 4	3	6	11	18	29	47
Svelte 5	2	4	7	11	24	37
Solid	2	4	7	11	24	35

4.2.5 Interleaved Batch Operations and Concurrent Updates

This benchmark assesses the performance of frameworks under conditions that simulate a realistic high-load environment, where multiple UI operations are interleaved and executed concurrently. In this scenario, operations such as additions, deletions, and updates are triggered simultaneously across a dashboard-like interface with many widgets. This stresses the framework's ability to schedule, batch, and coalesce updates effectively.

This measures the cumulative execution time required to process a batch of concurrent operations. We vary the batch size—testing with 10, 50, 100, 200, and 500 operations—under a simulated 2x CPU slowdown. This setup exposes the overheads associated with coordinating multiple simultaneous updates. Fine-grained reactive systems are expected to isolate updates to only the affected elements, whereas full re-rendering frameworks may incur extra overhead by re-executing larger portions of the component tree.

Table 4.8: Execution time (ms) for processing interleaved batch operations and concurrent updates (2x CPU slowdown)

Framework	Ops=10	Ops=50	Ops=100	Ops=200	Ops=500	Ops=1K
Angular	11	34	56	86	119	144
Angular (with Signals)	9	16	28	36	49	65
React	9	18	49	75	125	192
React (with Compiler)	9	17	35	63	98	132
Vue	5	11	24	32	45	66
Vue vapor	3	6	16	17	25	30
Svelte 4	6	10	15	28	60	71
Svelte 5	2	4	7	11	16	24
Solid	2	4	8	10	14	21

Chapter 5

Performance Benchmarks: A Comparative Analysis

This chapter presents a comparative analysis of CPU-centric benchmarks designed to evaluate the performance of different front-end rendering paradigms: dirty checking (represented by Angular), virtual DOM (React 19, Vue 3), and fine-grained reactivity (Solid, Svelte v4/v5, Vue Vapor, Angular 19 with Signals). We further analyze the impact of significant architectural shifts within frameworks, focusing on Svelte's evolution with runes (v4 to v5), Vue's Vapor mode, React's compiler, and Angular's adoption of signals. The primary focus is on script execution time, isolating framework overhead from browser rendering (layout, paint). This provides a precise measure of each approach's inherent update efficiency.

5.1 Methodology and Goals (Brief Recap)

As detailed in previous sections, the benchmarks leverage the Chrome Devtools Protocol (CDP) on Chrome version 131 to measure script execution time with millisecond-level precision. Tests were run on a Mac Book Air 13 with M3 chip 8-core CPU, 16 GB of RAM, using Puppeteer for CDP interaction. Each benchmark was executed 10 times per framework/version, with 5 warm-up runs. The mean and standard deviation of the 10 runs are reported. Benchmarks cover a range of scenarios: static/dynamic content creation (flat and tree structures), incremental list updates, table updates with dependencies, tree updates (root, internal, leaf), deeply nested dependency updates, and interleaved/concurrent operations.

5.2 Comparative Analysis of Rendering Paradigms

We first compare the three core rendering approaches: dirty checking, virtual DOM, and fine-grained reactivity, across the various benchmark scenarios.

5.2.1 Creation Overhead: Static and Dynamic Content

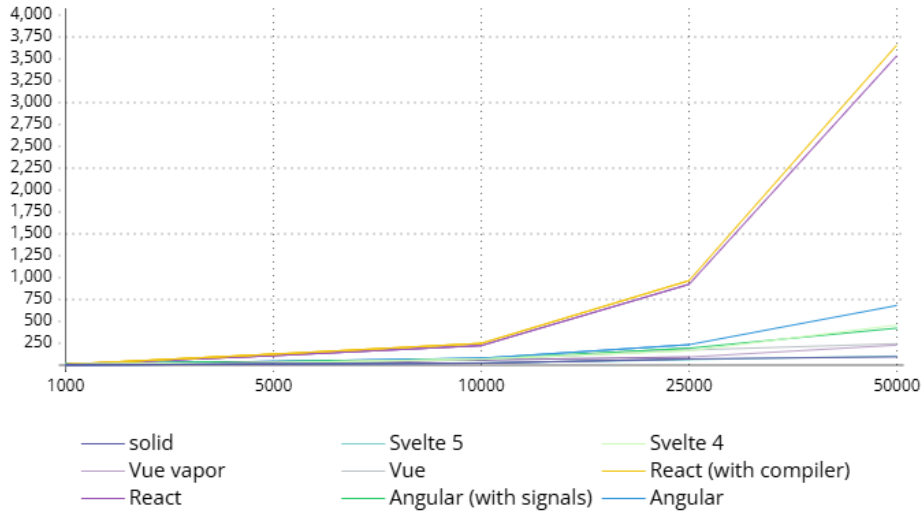

Figure 5.1: Script execution time (ms) for creating N static components, where N is the number of components.

Figure 5.1 compares creation times for static content.It presents a flat structure (N elements as direct children of a single parent). A clear performance hierarchy is observed: fine-grained reactive frameworks (Solid, Svelte, Vue Vapor) consistently outperform virtual DOM frameworks (React, Vue 3). This indicates a baseline overhead associated with the virtual DOM's reconciliation process, present even without DOM changes. Angular's Zones-based change detection introduces the highest overhead. The nonlinearity observed, particularly in the flat structure, suggests inefficiencies (possibly related to array iteration or DOM manipulation overhead) when handling large numbers of direct children.

Figure 5.2 shows creation times for dynamic components structured as a binary tree. While the performance gap between fine-grained reactive and virtual DOM approaches persists, the relative differences are potentially smaller. This is likely due to the added cost of component creation and dynamic binding, which impacts all frameworks. Component creation, even for simple components, introduces a non-negligible overhead. Preliminary analysis suggests this overhead *could* be

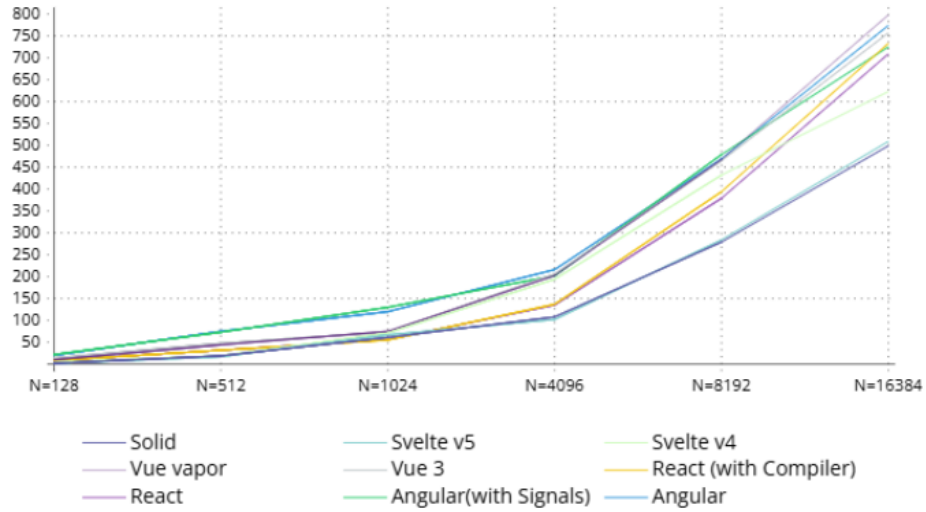

Figure 5.2: Script execution time (ms) for creating N dynamic components (binary tree).

roughly triple the cost of rendering equivalent static content directly within a parent component. However, it's important to note that real-world components typically manage a significantly larger number of elements, making the per-component overhead relatively less impactful in practice.

Framework	Script Execution Time (ms)	Full Render Cycle Time (ms)
Angular	120	368
React	56	475
Vue	75	346
Vue Vapor	75	322
Svelte 4	71	281
Svelte 5	67	423
Solid	61	274

Table 5.1: Comparison of script execution time and full render cycle time for creating N=1024 dynamic components (binary tree).

To provide a more complete picture of rendering costs, Table 5.1 shows script execution times with full render cycle times (including browser layout, painting, and compositing) for the dynamic binary tree creation scenario. The relative differences in full render cycle times are noticeably smaller than those observed in script execution times alone. This is expected because the components are relatively simple, and the browser's rendering pipeline (DOM construction, layout, painting) contributes a significant, relatively consistent overhead across all frameworks. This highlights that while framework choice significantly impacts *script* performance, the overall user-perceived rendering time is also heavily influenced by browser operations.

5.2.2 Performance in Incremental Updates: Precision and Efficiency

Figure 5.3 vividly illustrates the performance gap when dealing with localized updates. In these scenarios, frameworks like Svelte 5 and Solid, leveraging fine-grained reactivity fully, truly shine. Their architecture allows them to pinpoint and update only the specific dynamic values within the list, resulting in minimal script execution time. This is in stark contrast to Virtual DOM-based frameworks. While React, especially with the React Compiler, attempts to optimize this recomputation by recognizing static content, it still fundamentally relies on a diffing process. This process, even when optimized, inevitably involves a broader scope of computation compared to the direct DOM manipulation employed by fine-grained reactive systems. Angular, with its change detection mechanisms, also exhibits a less

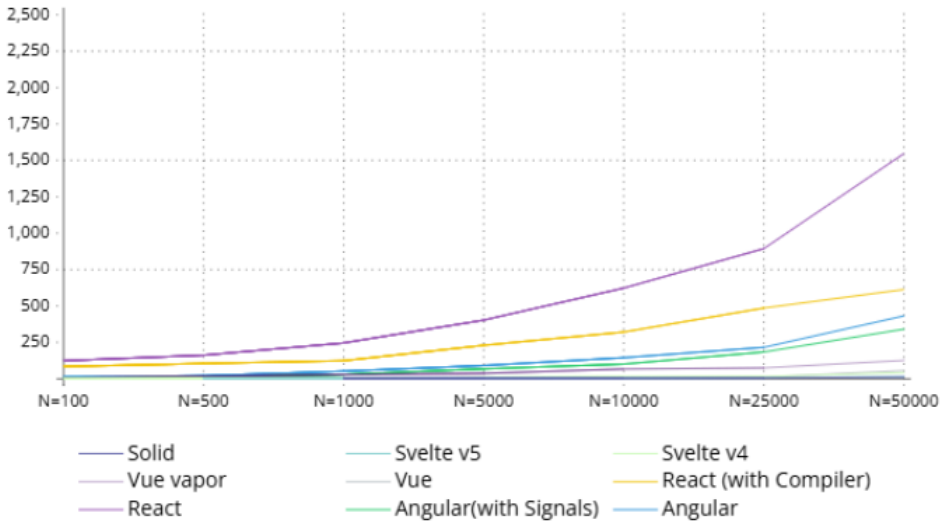

Figure 5.3: Script execution time (ms) for updating the root of a static flat list.

efficient update strategy compared to Svelte 5 and Solid in this benchmark.

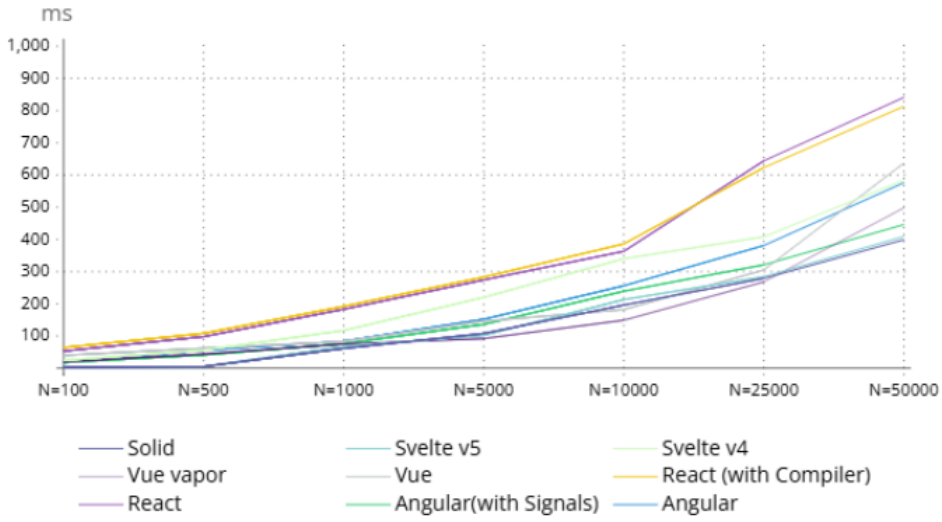

Figure 5.4: Script execution time (ms) for updating all elements of a flat list.

However, it's important to note that the performance gap between these approaches can narrow under different conditions. For instance, when performing mass updates, such as operations like "add one" to every item or "delete all" in the

list, the overhead associated with Virtual DOM diffing becomes less pronounced relative to the cost of individually updating a large number of elements in fine-grained systems. In such cases, the time spent reconstructing the Virtual DOM tree becomes comparable to the cumulative time spent updating each DOM node separately. Despite this, frameworks like Svelte 5 and Solid consistently demonstrate remarkable efficiency in scenarios demanding incremental updates, which are highly representative of real-world application behaviors where selective and localized DOM manipulations are paramount. This is further reinforced with the root and leaf of tree updates in Figure 5.5, where the complexity of such updates is constant against a $O(n)$ complexity with the vDOM and dirty checking approaches.

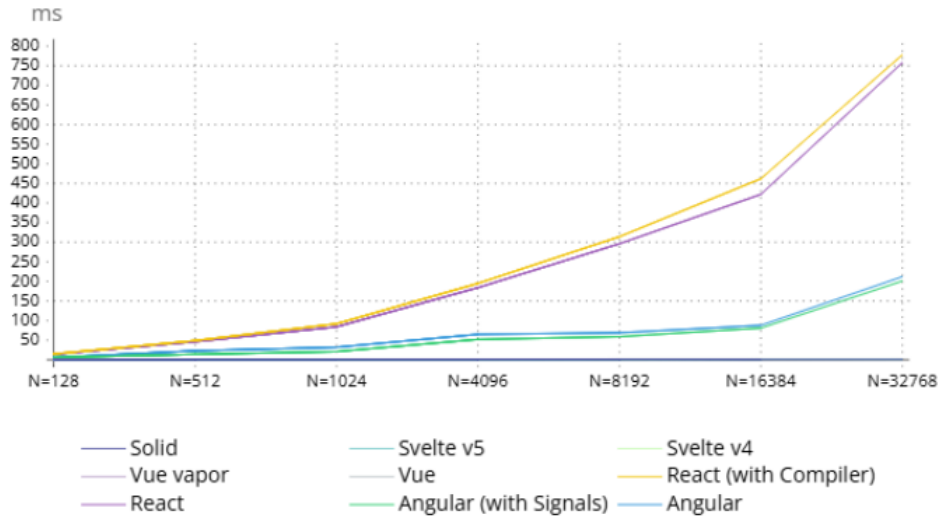

Figure 5.5: Script execution time (ms) for updating the root of a tree of N components.

Figure 5.6 (deeply nested dependency updates) highlights the importance of efficient dependency tracking. The results show that, contrary to initial expectations, the performance differences are minimal. Both the fine-grained and full re-render paradigms effectively batch updates, resulting in execution times that are nearly equivalent. This indicates that the overhead associated with explicitly traversing a dependency chain is comparable to that of rebuilding a subtree in a batched update scenario.

5.2.3 Concurrency and Complex Operations

Figure 5.7 (interleaved operations and concurrent updates) simulates a high-load scenario. Fine-grained frameworks, with efficient batching and scheduling, generally

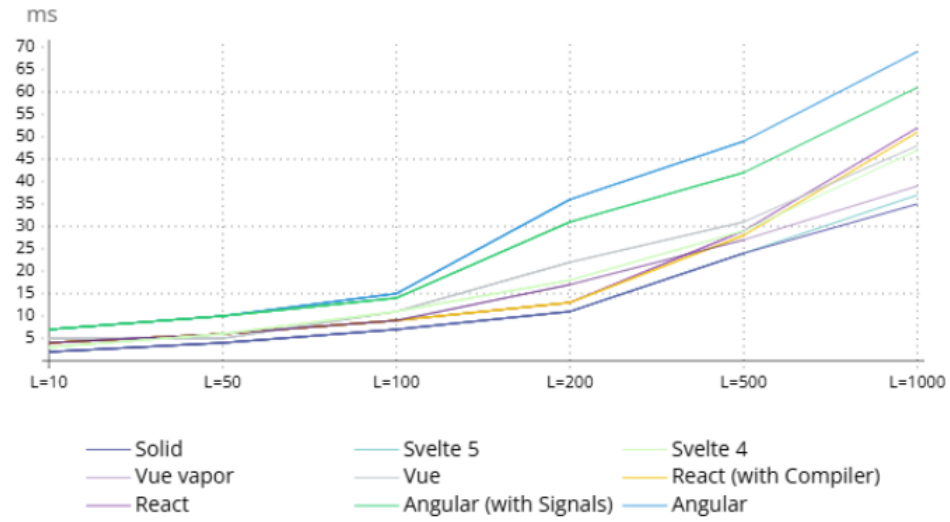

Figure 5.6: Script execution time (ms) for deeply nested dependency updates.

maintain responsiveness better than virtual DOM frameworks, where the overhead of diffing can become a bottleneck under pressure.

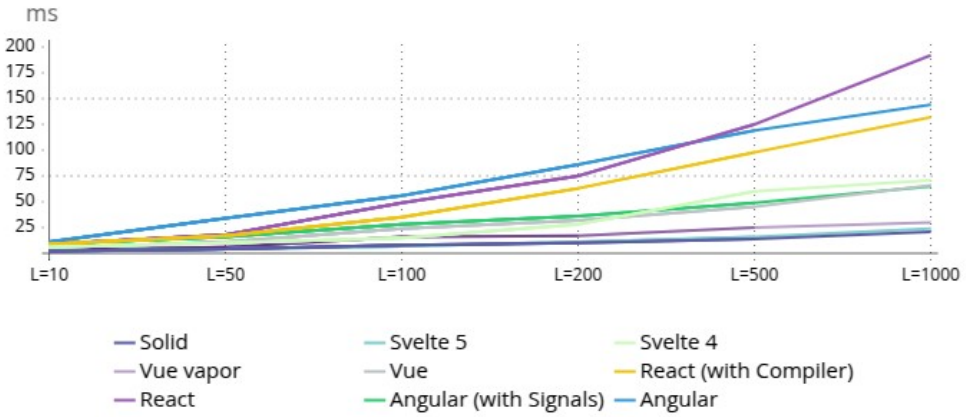

Figure 5.7: Script execution time (ms) under interleaved operations and concurrent updates.

More interestingly, frameworks that take advantage of the fine-grained approach in parallel with another system (like Svelte 4 with the compiler and Vue with their vDOM compilation step) see some inconsistencies in high load, primarily due to fixed costs associated with these approaches.

5.3 Intra-Framework Architectural Shifts

This section analyzes the performance impact of significant architectural changes within individual frameworks, using bar graphs to visually represent the before-and-after differences.

5.3.1 React

The integration of a compiler within the React framework has demonstrably enhanced its rendering efficiency by effectively recognizing and managing static content, thereby mitigating unnecessary recomputations. This improvement is distinctly illustrated in Figure 5.8, which highlights the substantial reduction in the computational cost associated with updating the root of a static, flat list compared to previous React versions lacking a compiler.

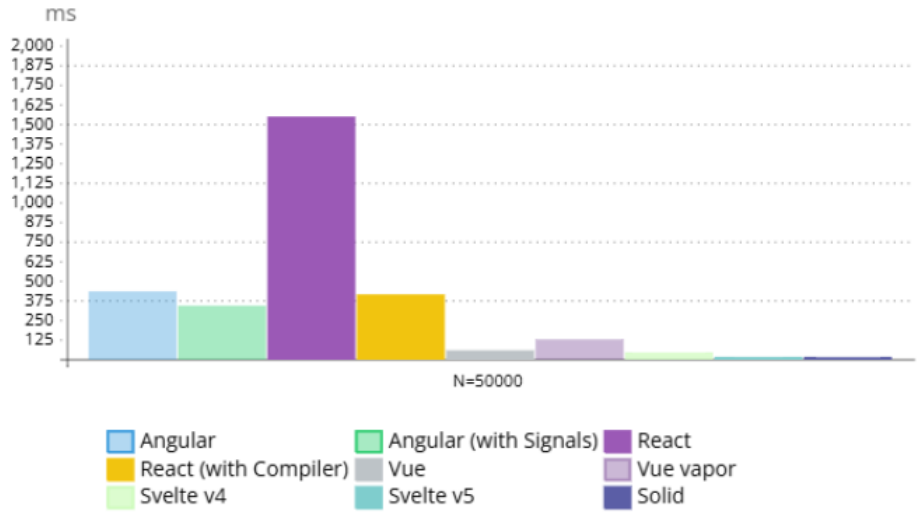

Figure 5.8: Script execution time (ms) for updating the parent of a flat list of N=10000 elements.

However, it is crucial to acknowledge that the performance gains attributed to the compiler are primarily confined to scenarios characterized by a significant proportion of static content. In situations where dynamic components predominate, the inherent limitations imposed by React's virtual DOM (vDOM) persist, and the performance differential between React and frameworks employing more fine-grained update mechanisms becomes evident.

5.3.2 Angular

The most dramatic shift is seen in Angular. The transition from Zone.js to Signals results in massive performance improvements across all benchmarks, especially in fixed costs. This clearly demonstrates the power of fine-grained reactivity over coarser-grained change detection.

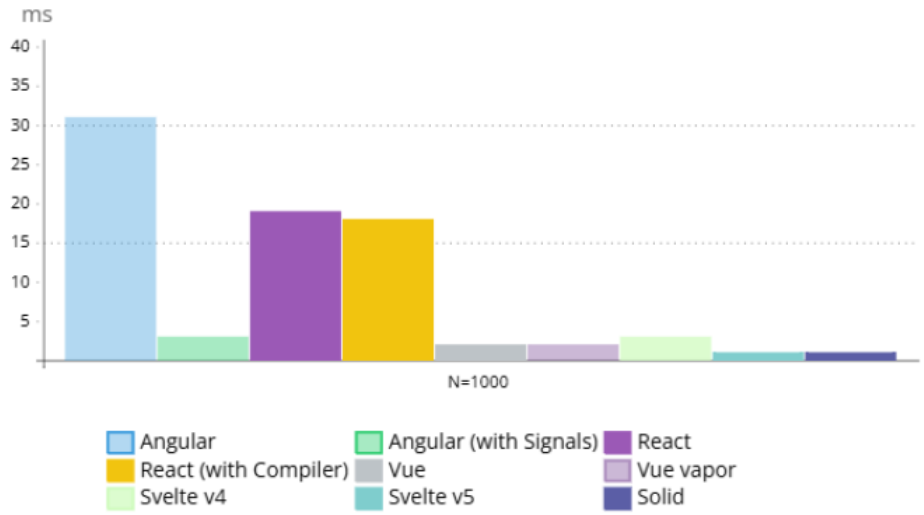

Figure 5.9: Script execution time (ms) for swapping two rows in flat list (3x CPU slowdown)

5.3.3 Svelte

Svelte's introduction of runes in v5 represents an evolution of its fine-grained reactivity. The benchmarks are expected to show modest but consistent improvements, as Svelte was already performing well but having some inconsistencies, indicating refinement of the underlying reactivity system. These inconsistencies are seen in effect under high load operations and Figure 5.10, where we see a clear performance boost under high load, effectively removing the fixed costs of the compiler from the previous version.

5.3.4 Vue

Moving from Vue 3 to Vue Vapor is demonstrating a significant increase in performance across a majority of the benchmarks as expected, due to removing the overhead associated with vDOM compilation. However, some inconsistencies still

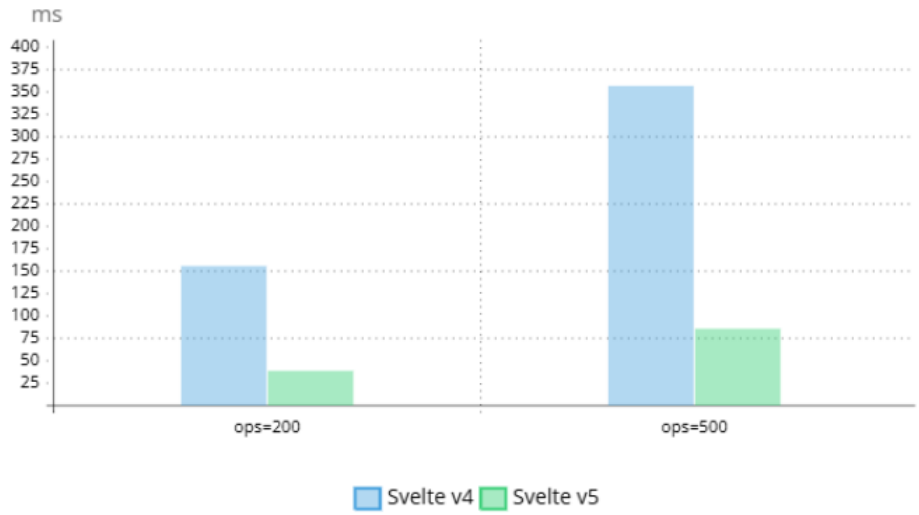

Figure 5.10: Script execution time (ms) for deeply nested dependency updates.

occur given the fact that it's still an experimental mode, but the results are rather positive overall.

5.4 Overall Conclusions

The benchmarks consistently demonstrate the performance advantages of fine-grained reactivity over both virtual DOM-based rendering and traditional dirty checking. Fine-grained systems excel in scenarios involving localized updates, complex dependency relationships, and high-frequency changes. Architectural shifts towards fine-grained reactivity (as seen in Angular) yield substantial performance gains. Compiler optimizations (as in React) can provide targeted improvements, but do not fundamentally alter the performance characteristics of the underlying rendering paradigm. The choice of front-end framework should be guided by the application's specific update patterns and performance requirements. For applications with frequent, localized updates and complex data dependencies, fine-grained reactive frameworks offer a compelling performance advantage.

Chapter 6

Contextualizing Benchmark Findings with Real-World Data

The experimental benchmark results presented in previous sections offer valuable insights into the performance characteristics of various frontend frameworks under controlled conditions. To contextualize these findings within real-world applications, this section analyzes HTTP Archive data comparing the performance of React, Angular, Vue.js, Svelte, and SolidJS across production websites. This analysis provides a complementary perspective to our controlled benchmark experiments and helps validate whether the observed performance patterns extend to real-world implementations.

6.1 Framework Adoption and Distribution

Before examining performance metrics, understanding the relative adoption of each framework provides important context for interpreting the data. Figure 6.1 illustrates the growth and relative market share of each framework from January 2020 to February 2025, based on HTTP Archive data [42].

As of February 2025, the data reveals significant disparities in framework usage:

- **React** maintains a dominant market position with approximately 1.1 million origins.
- **Vue** follows as the second most popular framework, with roughly 490,000 origins.
- **Angular** maintains a steady presence with about 108,750 origins.

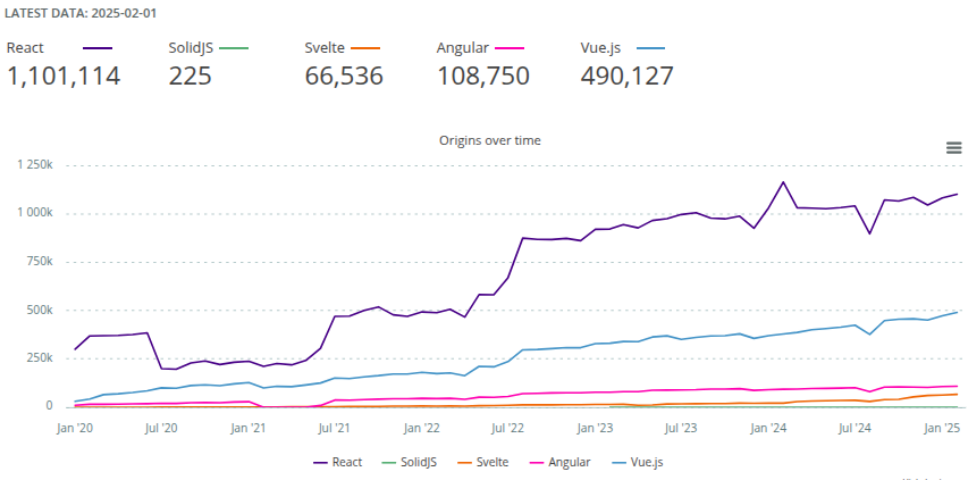

Figure 6.1: Origins over time for React, SolidJS, Svelte, Angular, and Vue. Data from [42].

- **Svelte**, though gaining traction with 66,536 origins, remains relatively niche compared to the market leaders.

- **Solid**, with only 225 origins, represents an emerging technology with minimal production adoption despite its promising performance characteristics.

The adoption patterns observed in HTTP Archive data confirm the varying levels of industry relevance for the frameworks included in our benchmark study, with React's dominance reflecting its widespread use in production environments.

6.2 Core Web Vitals Performance

The HTTP Archive provides valuable insights into how websites built with different frameworks perform against Google's Core Web Vitals metrics[43], which serve as important indicators of real-world user experience.

6.2.1 Time to First Byte (TTFB)

Figure6.2 shows the percentage of sites achieving a good Time to First Byte (TTFB) score[44].

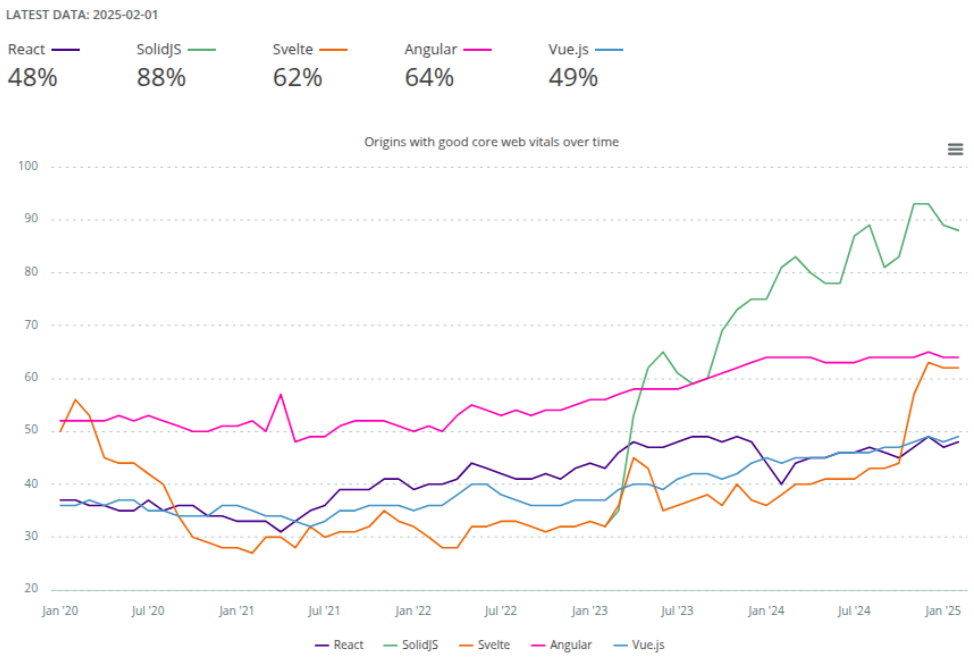

Figure 6.2: Percentage of sites achieving good TTFB scores. Data from [42].

TTFB measures the time between the request for a resource and when the first byte of a response begins to arrive, with values below 800 ms considered "good." Analysis of HTTP Archive data (see Figure 6.2) reveals:

- **SolidJS** significantly outperforms other frameworks in this metric, with 88% of SolidJS sites achieving good TTFB scores.
- **Angular** and **Svelte** follow with 64% and 62%, respectively.
- **React** (48%) and **Vue.js** (49%) show noticeably lower performance.

The superior TTFB performance of SolidJS sites aligns with our benchmark findings regarding its efficient rendering mechanism, though the small sample size (225 origins) should be considered when interpreting these results.

6.2.2 First Contentful Paint (FCP)

Figure6.3 shows the percentage of sites achieving a good First Contentful Paint (FCP) score[45].

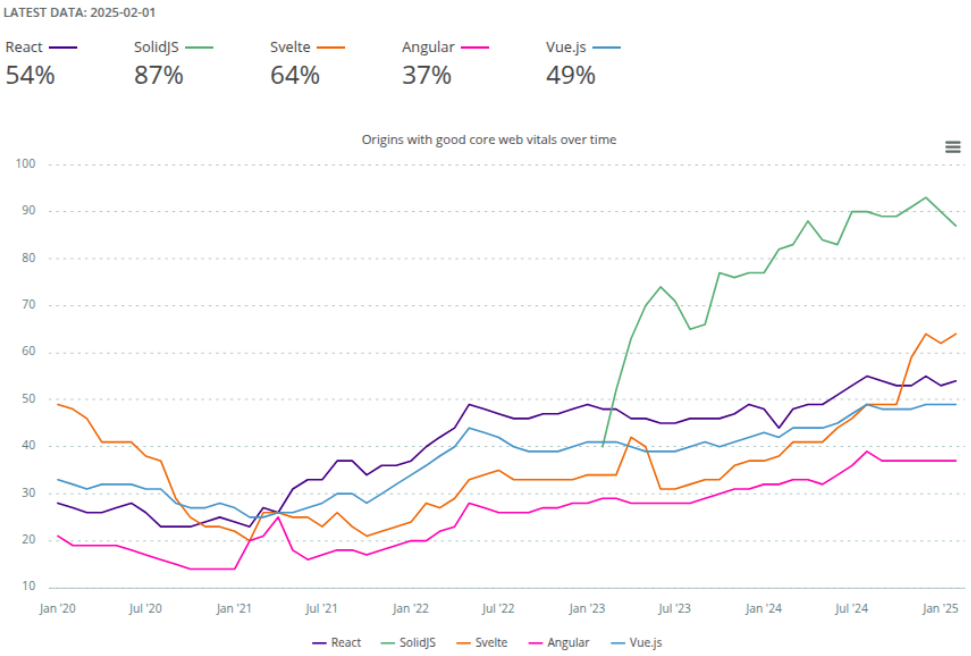

Figure 6.3: Percentage of sites achieving good FCP. Data from [42].

FCP[45] measures the time from when the page starts loading to when any part of the page's content is rendered on the screen, with good experiences defined as ≤ 1.8 seconds. Similar to TTFB, SolidJS leads with 87% of sites achieving good

FCP scores. Svelte follows at 64%, while React sits at 54%. Vue.js maintains a moderate 49% of sites with good FCP, while Angular lags behind at only 37%.

6.2.3 Largest Contentful Paint (LCP)

Figure6.4 shows the percentage of sites achieving a good Largest Contentful Paint (FCP) score[46].

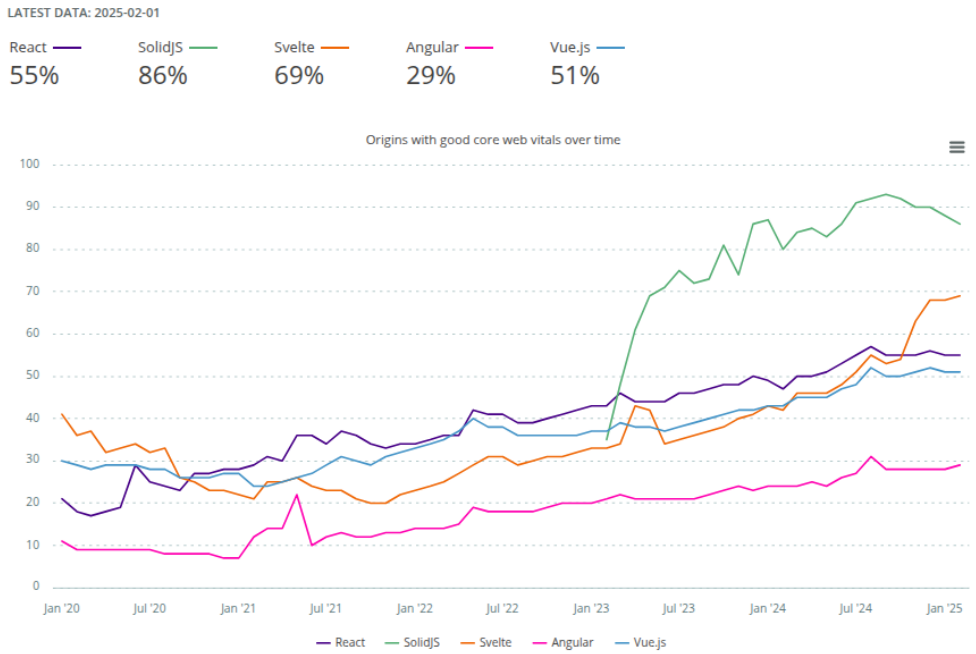

Figure 6.4: Percentage of sites achieving good LCP scores.Data from [42].

LCP measures when the largest content element in the viewport becomes visible, with good experiences defined as occurring within 2.5 s of page load. The HTTP Archive data shows SolidJS maintaining its performance lead with 86% of sites achieving good LCP scores. Svelte follows at 69%, while React (55%) and Vue.js (51%) show moderate performance. Angular significantly underperforms in this metric with only 29% of sites achieving good LCP scores.

6.2.4 Cumulative Layout Shift (CLS)

Figure6.5 shows the percentage of sites achieving a good Cumulative Layout Shift (CLS) score[47].

CLS quantifies unexpected layout shifts during a page's lifespan. Good scores are typically ≤ 0.1. Figure 6.5 (placeholder) shows that React achieves around 73%

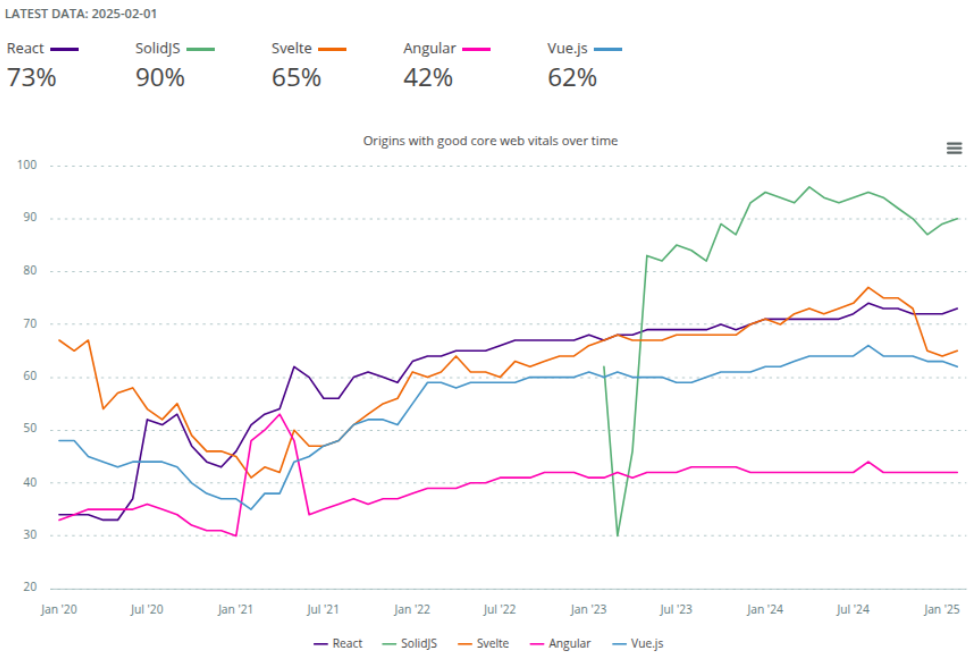

Figure 6.5: Percentage of sites achieving good CLS scores. Data from [42]

of sites with good CLS, while SolidJS (65%), Svelte (42%), Angular (49%), and Vue.js (47%) vary more widely. These differences may relate to how each framework handles hydration and dynamic updates, though developer implementation details can also heavily influence layout shifts in practice.

6.3 Correlation with Benchmark Findings

Our controlled benchmark experiments focused on metrics including script execution time, component creation efficiency, and update performance across the five frameworks. The HTTP Archive data provides an opportunity to validate whether these controlled measurements translate to real-world performance differences.

The superior performance of SolidJS and Svelte across Web Vitals metrics in HTTP Archive data corresponds with their excellent performance in our benchmark suite, particularly in scenarios involving component creation (Figures 5.1 and 5.2) and incremental updates (Figures 5.3 and 5.5). This suggests that the performance advantages of fine-grained reactivity observed in controlled environments do translate to measurable benefits in production applications.

The moderate performance of React and Vue in HTTP Archive data mirrors their mid-tier performance in our benchmark suite, particularly in scenarios involving complex component rendering and state updates (thus the move towards viper mode). Angular's underperformance in LCP metrics also corresponds with our finding that it exhibited higher overhead in component-heavy scenarios, though our benchmarks with Angular Signals showed significant improvements that may not yet be reflected in the HTTP Archive data given the recency of this architectural shift.

6.4 Limitations and Considerations

Several factors warrant caution when interpreting these HTTP Archive findings in relation to our benchmark results:

- **Sample Size Disparity**: The vast difference in sample sizes (1.1 million React origins vs. 225 Solid origins) introduces potential selection bias.
- **Implementation Variations**: Real-world implementations vary significantly in complexity, with many sites using multiple frameworks or partial implementations.
- **Network and Server Factors**: Web Vitals are influenced by factors beyond the framework itself, including network conditions, server infrastructure, and CDN usage—variables controlled for in our benchmark environment.

- **Version Distribution**: The HTTP Archive data encompasses all versions of each framework, while our benchmarks focused on specific, often latest, versions.
- **Developer Expertise**: The performance of real-world applications is significantly influenced by developer implementation choices that may not align with framework best practices.

6.5 Implications for Framework Selection

This correlation between our benchmark findings and real-world HTTP Archive data strengthens the case for considering rendering strategy as a factor in framework selection. The data suggests that fine-grained reactive frameworks like Solid and Svelte offer tangible performance benefits in production environments, not just in synthetic benchmarks.

However, the dominant market positions of React and Vue.js, despite their moderate performance metrics, underscore that factors beyond raw performance—such as ecosystem maturity, developer familiarity, and corporate backing—significantly influence framework adoption decisions.

Framework selection should thus balance performance considerations with other practical factors. For applications where performance is paramount, particularly those targeting mobile users or markets with limited connectivity, the data suggests that fine-grained reactive frameworks merit serious consideration despite their lower adoption rates.

Chapter 7

Conclusion

7.1 Summary of Findings

The empirical evidence from this study firmly establishes the performance supremacy of fine-grained reactive frameworks—exemplified by Solid, Svelte, Vue Vapor, and Angular 19 with Signals—over both legacy dirty-checking mechanisms (Angular 15) and virtual DOM-centric architectures (React 19, Vue 3). Across a spectrum of benchmark scenarios, encompassing static rendering, intricate dynamic updates, and concurrent operations, fine-grained reactivity consistently demonstrated superior script execution times, frequently by substantial margins. Solid and Svelte, frameworks built from the ground up embracing this paradigm, emerged as clear frontrunners in performance.

This performance edge is intrinsically linked to the granular update strategy inherent in fine-grained reactivity. By meticulously tracking dependencies at the level of individual data bindings, these systems enable highly targeted DOM manipulations. Consequently, only precisely impacted DOM nodes undergo re-rendering upon data mutation, circumventing the performance tax associated with broader change detection cycles or the virtual DOM's diffing and reconciliation processes.

The performance gains observed in Angular's transition to Signals, and Vue's adoption of Vapor mode, underscore a significant architectural trend across frameworks: a move towards fine-grained reactivity to achieve optimal performance. These advancements, alongside Svelte's continuous refinement, indicate a clear industry-wide recognition of the benefits of this approach. React, with its continued reliance on the virtual DOM and an optional compiler for performance enhancement, stands as a notable outlier in this landscape, pursuing a different strategy amidst the growing consensus around fine-grained reactivity.

7.2 Implications for Future Frontend Development

The substantial performance advantages of fine-grained reactivity, alongside its growing adoption throughout the frontend ecosystem, signal a transformative shift in web UI development. The trajectory clearly points toward frameworks that emphasize computational efficiency and minimize DOM manipulations by fundamentally embracing fine-grained reactivity. Prominent frameworks aren't merely adopting this approach but actively advocating for its incorporation into the core language specifications of the web platform itself, indicating a profound commitment to this paradigm. While the performance benefits of fine-grained reactivity are compelling, and increasingly viewed as essential for competitive frameworks, React maintains a distinct philosophical position centered on developer experience and a streamlined data flow model. React's fundamental principle is to handle initialization and updates uniformly, abstracting away the complexities of reactive primitives and emphasizing a unidirectional data flow. This "no-thinking-about-data-flow" methodology prioritizes developer intuition and code readability, potentially sacrificing inherent runtime performance.

React's approach relies on an opt-in compiler to address the performance gap, seeking to automatically inject fine-grained optimizations into its virtual DOM-based rendering process. While this compiler delivers performance improvements, it remains uncertain whether it can entirely eliminate the performance differential with frameworks designed from first principles for fine-grained reactivity. The long-term consequences of this divergent strategy represent an area of ongoing observation and industry discourse. It is particularly noteworthy to consider how this strategic distinction will evolve as web applications become progressively complex and performance-critical

7.3 Compilers and AI's Potential Influence

The increasing sophistication of front-end frameworks, particularly the emergence of compiler-driven optimizations, indicates a fundamental transformation in web development methodologies. Contemporary frameworks are evolving beyond their traditional role as runtime libraries into complex optimizing compiler systems. The JavaScript code authored by developers is progressively becoming a higher-level abstraction, with browsers ultimately executing highly optimized, compiler-generated instructions.

This evolutionary trajectory exhibits notable parallels with native application development paradigms, where compiled binaries undergo extensive optimization processes that substantially diverge from the original source code. Similarly, web

assets will increasingly result from sophisticated build processes, facilitating aggressive optimizations such as fine-grained reactivity enhancements that would be computationally prohibitive in manually authored JavaScript. This progression raises significant questions regarding the future of web platform transparency principles, particularly the "View Source" accessibility that has historically characterized the web ecosystem. Nevertheless, the quantifiable user experience improvements, especially on computationally constrained devices, present compelling justification for these architectural shifts.

From a forward-looking perspective, the emergence of Artificial Intelligence represents a potentially significant variable in framework evolution. While computational performance remains a critical metric, the conceptual simplicity of React's unidirectional data flow architecture and its relatively imperative-style programming model may provide substantial advantages in the context of AI-assisted code generation. Large Language Models (LLMs) potentially demonstrate enhanced capabilities in generating and maintaining code within simpler, more deterministic programming frameworks like React's, even when such frameworks necessitate compiler-based optimizations to achieve optimal runtime performance. In this emerging computational landscape, React's strategic emphasis on developer conceptual simplicity coupled with compiler optimization could prove remarkably prescient, effectively aligning with the evolving paradigm of AI-augmented software development methodologies

7.4 Limitations of the Study

This study, while providing valuable insights, is subject to certain limitations:

- Simplified Benchmarks: The benchmarks, designed for targeted performance analysis, offer simplified representations of complex real-world applications, potentially overlooking nuances introduced by intricate interactions or third-party libraries.

- Specific Framework Versions: The study's focus on specific framework versions means that future updates and optimizations could alter the observed performance landscape.

- Single Browser: Benchmarking was conducted using the Chrome Devtools Protocol, and performance characteristics may exhibit variations across different browsers and rendering engines.

- Hardware Dependency: Results are inherently influenced by the specific hardware configuration used for testing.

7.5 Recommendations for Further Research

Future research avenues include:

- Real-World Application Benchmarks: Developing benchmarks based on more complex, real-world applications to evaluate performance under realistic conditions.
- Cross-Browser Comparisons: Expanding benchmarks to encompass major browsers (Firefox, Safari, Edge) for cross-browser performance analysis.
- Compiler Optimization Analysis: In-depth investigation of the specific optimizations implemented by React's compiler and their effectiveness across diverse code patterns.
- Memory Usage Analysis: Evaluating memory consumption alongside script execution time, as memory management is crucial for application performance.
- Long-Term Performance: Studying performance trends over time as applications scale in size and complexity.
- AI-Assisted Code Generation: Exploring the efficacy of AI in generating performant frontend code, particularly in the context of different framework paradigms.

Appendix A

Frameworks

Table A.1: The list of the frameworks versions used in for the benchmarks with thier respective bundler

Framework	Version	Bundler
Angular	15.2.10	Webpack
Angular(with Signals)	19.0.3	Vite
React	19.0.0	Vite
React (with Compiler)	19.0.0	Vite
Vue 3	3.5.3	Vite
Vue vapor	3.2.2024-761f785	Vite
Svelte v4	4.2.17	Rollup
Svelte v5	v5.0.5	Vite
Solid	1.8.15	Vite

Bibliography

[1] Matija Varga. «The evolution of web browser architecture». In: (Jan. 2013), pp. 569–571 (cit. on p. 3).

[2] W3.org. «CSS Object Model (CSSOM)». In: W3.org, 2021 (cit. on p. 5).

[3] «The Rendering Critical Path». In: *https://www.chromium.org/developers/the-rendering-critical-path/* (2009) (cit. on p. 6).

[4] Harini Natarejan. «Improving a website's first meaningful paint by optimizing render blocking resources». MA thesis. Malmo School of Technology, 2017 (cit. on p. 6).

[5] Jon Duckett. *JavaScript and jQuery: Interactive Front-End Web Development*. 2nd. Wiley, 2014. ISBN: 9781118531648 (cit. on p. 8).

[6] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, 2012. ISBN: 9781449331818 (cit. on p. 8).

[7] Addy Osmani. *Developing Backbone.js Applications*. O'Reilly Media, 2012 (cit. on p. 8).

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. ISBN: 9780201633610 (cit. on p. 8).

[9] *How MVC Works: Controller Flow*. `https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/what-is-mvc`. Accessed: Feb 14, 2025. 2010 (cit. on p. 9).

[10] R. Kumar and A. Sharma. «Implementing the MVC Pattern in Client-Side Web Frameworks: A Case Study of Backbone.js». In: *Proceedings of the IEEE/ACM International Conference on Web Engineering*. 2016, pp. 115–122 (cit. on p. 9).

[11] Martin Fowler. «Architectural Patterns: Model–View–Controller». In: *IEEE Software* 21.5 (2004), pp. 56–65. DOI: `10.1109/MS.2004.1318370` (cit. on p. 10).

[12] E. J. Smith and R. T. Brown. «Event-Driven Architectures in MVC Frameworks: A Comparative Study». In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. 2018, pp. 212–220 (cit. on p. 10).

[13] C. F. Ocariza and K. Bajaj. «Imperative DOM Manipulation: Sources of Errors in Web Applications». In: *Proceedings of the IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 55–64 (cit. on p. 10).

[14] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. «An empirical study of client-side JavaScript bugs». In: *IEEE International Symposium on Empirical Software Engineering and Measurement* 23 (2013), pp. 55–64 (cit. on p. 10).

[15] M. D. Smith and L. K. Johnson. «A Formal Analysis of State Explosion in Modern User Interfaces». In: *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. 2018, pp. 89–96 (cit. on p. 10).

[16] A. Kumar and P. Lee. «Declarative Versus Imperative Programming Paradigms: A Comparative Study in Web Development». In: *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*. 2019, pp. 102–110 (cit. on p. 11).

[17] J. Chen and S. Liu. «Analyzing the Performance Overhead of DOM Reconciliation in Declarative UI Frameworks». In: *Proceedings of the ACM Symposium on Web Performance (WebPerf)*. 2021, pp. 37–44 (cit. on p. 11).

[18] *Reconciliation – React*. Accessed: Feb 19, 2021. 2021. URL: `https://reactjs.org/docs/reconciliation.html` (cit. on pp. 11, 15).

[19] Shyam Seshadri and Brad Green. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps*. O'Reilly Media, 2014 (cit. on p. 11).

[20] Alice Smith. «Evaluating Declarative Data Binding Mechanisms in Modern Web Frameworks». PhD thesis. Stanford University, 2016 (cit. on p. 11).

[21] Michael J. Hart. «From MVC to MVVM: Evolving Architectural Patterns in Modern Web Applications». In: *IEEE Software* 35.3 (2018), pp. 26–32. DOI: `10.1109/MS.2018.2800456` (cit. on p. 11).

[22] *React: A JavaScript library for building user interfaces*. Accessed: Feb 14, 2025. 2025. URL: `https://reactjs.org/` (cit. on pp. 12, 18).

[23] Alex Banks and Eve Porcello. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, 2017. ISBN: 9781491954621 (cit. on p. 12).

[24] John Gossman. «The Model-View-ViewModel (MVVM) Design Pattern». In: *Proceedings of the IEEE Conference on Software Architecture (ICSA)*. Discusses the evolution and practical aspects of MVVM in modern application design. 2009, pp. 461–499 (cit. on p. 12).

[25] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2002 (cit. on p. 13).

[26] Cassio de Sousa Antonio. *Pro React*. Apress, 2016 (cit. on p. 13).

[27] *Introducing JSX*. Accessed: Feb 19, 2021. 2021. URL: https://reactjs.org/docs/introducing-jsx.html (cit. on p. 17).

[28] R. Zhuykov and E. Sharygin. «Ahead-of-Time Compilation of JavaScript Programs». In: *Programming and Computer Software* (2017). DOI: 10.1134/S036176881701008X (cit. on p. 17).

[29] Roland Kuhn, Jamie Allen, and Brian Hanafee. *Reactive Design Patterns: Best Practices for Building Resilient, Event-Driven Applications*. Manning Publications, 2017 (cit. on p. 17).

[30] Umut Acar, Guy E. Blelloch, and Robert Harper. «Self-Adjusting Computation». In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. 2006, pp. 355–364. DOI: 10.1145/1134277.1134318 (cit. on p. 17).

[31] *Virtual DOM and Internals – React*. Accessed: May 04, 2021. 2021. URL: https://reactjs.org/docs/faq-internals.html (cit. on p. 18).

[32] Lauren Tan. «React Compiler Beta Release». In: *React Dev Blog* (Oct. 2024). URL: https://react.dev/blog/2024/10/21/react-compiler-beta-release (cit. on p. 19).

[33] *SolidJS: Declarative JavaScript library for building user interfaces*. Accessed: Feb 14, 2025. 2025. URL: https://solidjs.com/ (cit. on p. 19).

[34] Bruno Couriol. «TC39 Proposal for Signals: Harmonizing Reactive Programming Primitives in JavaScript». In: *InfoQ* (2024), pp. 1–4 (cit. on p. 20).

[35] *Vue.js: The Progressive JavaScript Framework*. Accessed: Feb 14, 2025. 2025. URL: https://vuejs.org/ (cit. on p. 22).

[36] *Angular: One framework. Mobile and desktop.* Accessed: Feb 14, 2025. 2025. URL: https://angular.io/ (cit. on p. 24).

[37] *Angular Signals*. Accessed: Feb 14, 2025. 2025. URL: https://angular.dev/guide/signals (cit. on p. 24).

[38] *Svelte: Cybernetically enhanced web apps*. Accessed: Feb 14, 2025. 2025. URL: https://svelte.dev/ (cit. on p. 26).

[39] Rich Harris and the Svelte Team. *Introducing runes*. Svelte 5 runes: a new syntax for reactive state via signals. 2023. URL: https://svelte.dev/blog/runes (cit. on p. 26).

[40] Stefan Krause. *krausest/js-framework-benchmark*. Accessed: May 26, 2021. 2021. URL: https://github.com/krausest/js-framework-benchmark (cit. on p. 28).

[41] *Chrome DevTools Protocol*. Accessed: May 26, 2021. 2021. URL: https://chromedevtools.github.io/devtools-protocol/ (cit. on p. 29).

[42] *HTTP Archive: Trends in Web Technology (February 2025)*. Accessed: Feb 14, 2025. 2025. URL: https://httparchive.org/reports/state-of-the-web?month=2025-02 (cit. on pp. 45–50).

[43] *Core Web Vitals – Essential metrics for a healthy site*. Accessed: Feb 14, 2025. 2025. URL: https://web.dev/vitals/ (cit. on p. 47).

[44] *Time to First Byte (TTFB)*. Accessed: Feb 14, 2025. 2025. URL: https://developer.chrome.com/docs/lighthouse/performance/ttfb/ (cit. on p. 47).

[45] *First Contentful Paint (FCP)*. Accessed: Feb 14, 2025. 2025. URL: https://web.dev/fcp/ (cit. on p. 48).

[46] *Largest Contentful Paint (LCP)*. Accessed: Feb 14, 2025. 2025. URL: https://web.dev/lcp/ (cit. on p. 49).

[47] *Cumulative Layout Shift (CLS)*. Accessed: Feb 14, 2025. 2025. URL: https://web.dev/cls/ (cit. on p. 49).