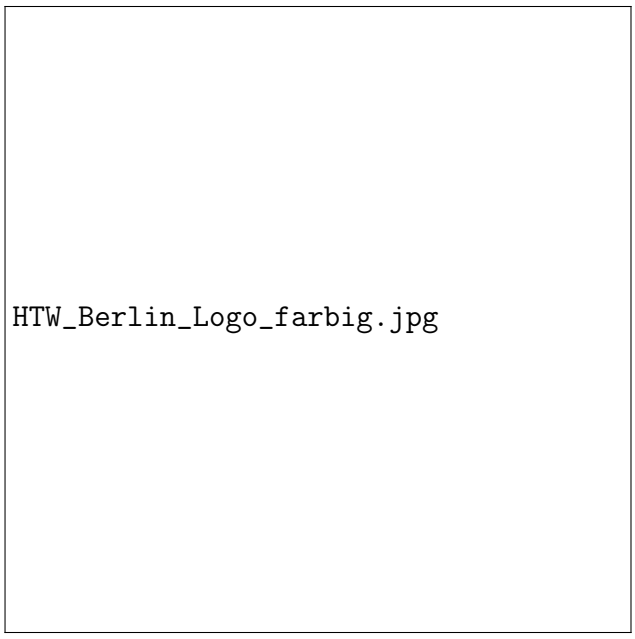

A Multi-Layered Visual Application Builder with Dynamic CSS Class Generation

Bachelor Thesis

submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Science (B.Sc.)

at

Berlin University of Applied Sciences
Faculty 4: Computer Science, Communication and Economics
Study Program *International Media and Computing*

First Supervisor: Academic Title First Name Last Name
Second Supervisor: Academic Title First Name Last Name

Submitted by: Your Full Name [Student ID]
Berlin, August 26, 2025

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Prof. Dr. [Name] and Prof. Dr. [Name], for their guidance and support throughout this research. I also thank [other people/institutions] for their valuable contributions.

Abstract

When faced with the challenge of creating content and interactive web experiences, less technical users have to rely on like WordPress and editors.

Although these tools partially satisfy the needs of modern web experiences, many have limitations regarding the they produce, and an inaccuracy between the preview layer and the final rendered content in the Web browser.

Due to the static output of tools like Webflow, WordPress, Wix, etc., users cannot make use of the features that come with modern JavaScript frameworks like React, NextJS or VueJS, which are based on the dynamic injection of JavaScript code during .

The following thesis answers the question of how a multilayered visual application builder allowing real-time editing, dynamic CSS class generation, and multi-touch interactions can be implemented using the modern JavaScript framework NextJS.

As a result, the barrier to build great software is lowered by applying meta-programming concepts.

Meta Programming is defined as follows:

"Meta-programs analyze, generate, and transform object programs. In this process, object programs are structured data."[1]

In the context of this thesis, it is used to implement a program that dynamically generates other programs.

In specific SPAs during , by using a component based approach with drag and drop functionality and real-time editing.

[1] **visser2002**.

Contents

List of Figures

List of Tables

Listings

1. Introduction

The development of content and interactive web experiences is a difficult task, especially for less technical users. In the early days of Web 1.0, developers worked with little more than HTML and CSS. Over time, however, the landscape grew increasingly complex. The proliferation of mobile computing devices—including smartphones, tablets, and wearables such as smartwatches—has significantly expanded the possibilities for digital interaction. However, it has also introduced intricate design challenges, notably the imperative for responsive design to accommodate diverse screen sizes and resolutions, as well as the complexities of multi-touch interactions.**kim2021designpatternstradeoffsresp**

To address the mounting complexity inherent in modern web development, such as Webflow, WordPress, and Wix have emerged. These platforms empower users to craft responsive and interactive web experiences without writing code. Leveraging visual interface builders with drag-and-drop component functionality, they abstract away the technical intricacies of responsive layout design and cross-device compatibility.

"An increasing number of software applications are being written by end users without formal software development training."**kuhail2021characterizing** This inspired large technology companies [...] to invest in low-code development environments empowering end users to create web and mobile applications"**kuhail2021characterizing**

With an annual growth rate of more than 20% s are expected to continue to grow up to a total market valuation of 50 billion USD by 2028.**bratincevic2024lowcode**

This clearly shows the need for tools that allow users to create web applications without having to write code.

1.1 Problem Statement

The central problem addressed in this thesis lies in reconciling two fundamentally different requirements in the design of visual application builders:

1. **Editor Reactivity**: Visual editors must provide immediate, synchronized feedback across multiple interface layers (e.g., component palette, canvas, layers panel, properties panel). User actions such as dragging components, modifying styles, or switching responsive breakpoints must instantly propagate to all relevant parts of the editor interface.
2. **Runtime Interactivity**: At the same time, components within the application must preserve their intended runtime behavior. Buttons must remain clickable, forms must handle input and validation, and interactive elements must reflect accurate state management. This behavior must remain coherent both in the editor preview and in the exported application.

Bridging these two reactive systems creates complex architectural constraints.

Existing tools such as Framer and Webflow demonstrate the viability of multi-layered visual builders but typically achieve this by limiting developer control over the underlying framework. As a result, the expressive capabilities of React, particularly its component model, state management, and runtime flexibility, are only partially preserved.

This thesis does not seek to fully replicate the entirety of React's capabilities within a visual builder. Its goal is to investigate to what extent real-time UI editing can coexist with React's runtime architecture, and to identify the architectural patterns and meta-programming strategies that make this possible. By doing so, it addresses the gap between highly abstract visual editors and the framework-level fidelity required for professional application development.

1.2 System Overview

In the following a very high level overview of the system is provided. A more detailed discussion of the individual layers and their interaction patterns is provided in the following chapters.

1. **Component Palette**: Drag-and-drop library of available UI components
2. **Preview Canvas**: Main editing area with desktop, tablet, and mobile viewports
3. **Layers Panel**: Hierarchical tree view of component structure and relationships
4. **Properties Panel**: Dynamic style and behavior editor for selected components

As demonstrated in Figure **??**, Framer already implements variations of this multi-layered approach, showing the practical viability of such interfaces for visual application development.

1.3 Research Questions, Scope, and Contributions

This thesis is guided by one overarching research question:

How can a multi-layered visual interface enable real-time application building while maintaining React interactivity through a meta-programming approach that transforms editor configurations into functional components?

Two supporting questions refine this investigation: (1) how can the four interface layers—the component palette, canvas, layers panel, and properties panel—remain synchronized in real time during visual editing operations, ensuring consistent editor reactivity? and (2) which architectural patterns enable the transformation of structured, serializable data into interactive React components?

The scope of the thesis is confined to the conceptual and technical exploration of architectural patterns and meta-programming approaches for a multi-layered visual application builder integrated with React. The goal is not to deliver a production-ready platform but to examine the feasibility of combining real-time editing with runtime interactivity, focusing on the tensions between editor

reactivity and component behavior at runtime.

To pursue these questions, four technical objectives are defined:

1. design of a unified component tree model that supports visual editing and React export
2. development of a responsive preview system for desktop, tablet, and mobile viewports
3. creation of a dynamic property editing interface adaptable to different component types
4. implementation of a mechanism to extend the component palette with custom components

The contributions of this research are both analytical and architectural. Analytically, it provides a structured problem analysis of the dual-reactivity challenge in visual application builders and offers a conceptual framework for organizing multi-layered interfaces around a unified component tree model. Architecturally, it identifies design patterns for transforming serialized editor configurations into interactive React components and examines the trade-offs between abstraction for accessibility and fidelity to the expressive power of React. Together, these contributions aim to advance the understanding of how real-time visual editing environments can be meaningfully combined with modern JavaScript frameworks, while also laying the foundation for future research and development in this domain.

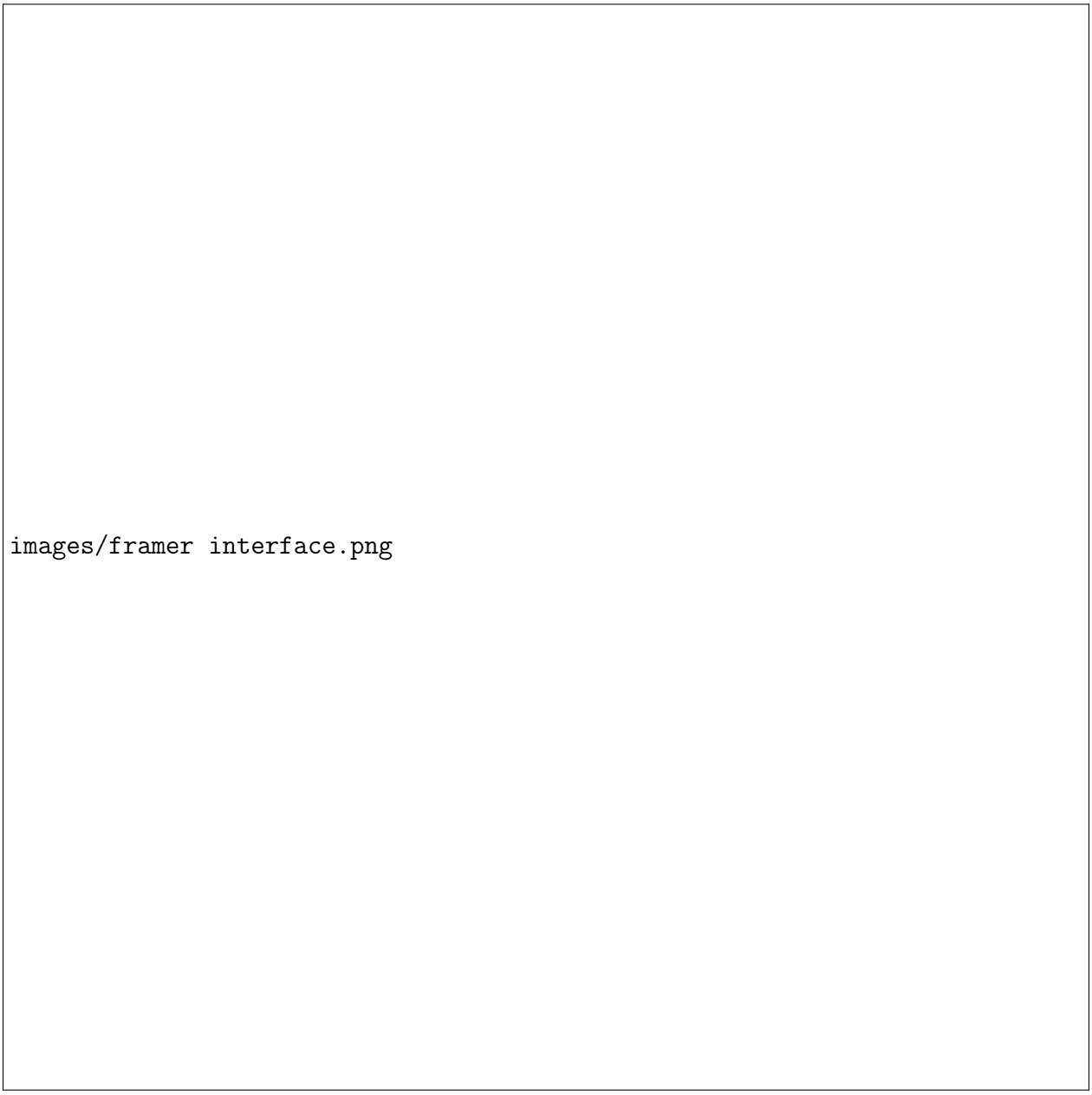

Figure 1.1: Framer Interface showing multi-layered visual editing environment with layers panel (left), responsive canvas (center), properties panel (right), and component palette (top - currently not shown)**framer**

2. Foundations

This chapter establishes the theoretical foundation necessary for understanding the multi-layered visual application builder. It encompasses domain knowledge about visual application development platforms, analysis of existing solutions and their architectural patterns, and the core technological concepts that enable runtime component generation and reactive state management.

The chapter begins with an examination of the visual application development landscape and the stakeholder groups served by these tools, providing essential context for understanding user requirements and system constraints.

This is followed by a comprehensive analysis of related work and existing solutions, identifying both successful patterns and limitations in current approaches. The chapter concludes with an exploration of relevant technological concepts, focusing on JavaScript manipulation, React's architecture, reactive programming paradigms, and meta-programming techniques that form the technical foundation for the proposed system.

2.1 Domain Knowledge and Context

Understanding the domain context is essential for positioning the multi-layered visual application builder within the broader ecosystem of development tools and user needs. This section reviews the current landscape of visual development platforms, outlining key solution categories and their limitations with respect to modern JavaScript frameworks. The analysis considers the rise of low-code platforms, the technical barriers faced by less-technical users, and the diverse stakeholder groups such builders must serve. These insights shape the architectural decisions and design priorities of the proposed system.

2.1.1 Visual Application Development Landscape

The landscape of visual application development has been fundamentally transformed by the emergence of . As comprehensively analyzed by Sahay et al. **sahay2020supporting**, these platforms represent a paradigm shift toward democratizing software development by reducing technical barriers through visual interfaces and model-driven approaches.

Traditional web development requires extensive technical knowledge of , , JavaScript, and modern frameworks like React or Vue.js. This technical barrier has historically limited content creators, designers, and non-technical users from building interactive web applications independently.

LCDPs address this challenge by providing interfaces that enable application creation through direct manipulation rather than code writing.

However, the current LCDP ecosystem reveals distinct categories with inherent architectural limitations:

Static Site Builders (WordPress, Godaddy, Squarespace): Generate static and output with limited interactivity and no true capabilities

Design-to-Code Tools (Webflow, Adobe XD): Excel in visual design fidelity but lack dynamic application logic and state management

Component-Based Builders (Framer, Bubble): Offer dynamic functionality through proprietary runtime systems, creating vendor lock-in challenges

The fundamental challenge identified by Sahay et al. lies in bridging the gap between visual design tools and modern JavaScript framework capabilities, particularly React's component-based architecture and dynamic state management.

2.1.2 Target User Groups and Stakeholders

Visual application builders serve multiple stakeholder groups with varying technical backgrounds and requirements:

- **Content Creators**: Need rapid and intuitive visual prototyping and landing page creation
- **Designers**: Require pixel-perfect visual control with responsive design capabilities
- **Developers**: Seek faster iteration cycles and visual debugging tools
- **Product Teams**: Need collaborative tools that bridge design and development workflows
- **Agencies**: Require scalable solutions for client work with quick turnaround times

Each group brings distinct requirements for interactivity levels, customization depth, and technical control, necessitating a flexible architecture that accommodates various use cases.

2.2 Related Work and Existing Solutions

This section examines existing visual application building platforms to understand current approaches, architectural patterns, and limitations. The analysis provides context for the proposed multi-layered approach and identifies gaps that this thesis addresses.

2.2.1 Analysis of Current Visual Building Platforms

The examination covers three categories of platforms: component-based visual development tools (Framer), design-centric approaches (Webflow), and code-first visual editing environments (Codux). Each platform represents different trade-offs between visual design capabilities, runtime interactivity, and technical flexibility.

2.2.1.1 Framer: Component-Based Visual Development

Framer represents the current state-of-the-art in visual application builders, combining design tools with React-based component rendering. Key architectural insights from Framer include:

- **Runtime Component System**: Rather than generating static code, Framer maintains a centralized data model that drives dynamic React component rendering
- **Live Preview Architecture**: Changes to the visual interface immediately reflect in a live preview through reactive state management
- **Component Tree Model**: Applications are represented as hierarchical component trees with serializable state
- **Override System**: Developers can inject custom logic through JavaScript "overrides" that modify component behavior

However, Framer's proprietary nature limits extensibility and creates vendor lock-in for users seeking to migrate to standard React applications.

2.2.1.2 Webflow: Design-Centric Approach

Webflow focuses primarily on visual design with limited dynamic functionality:

- **Static Output**: Generates traditional and with minimal JavaScript interactivity
- **Animation System**: Built-in animation tools but no true state management
- **CMS Integration**: Content management capabilities but no dynamic application logic

While Webflow excels in design fidelity, it cannot produce modern applications with complex state management and component interactions. It is rather focused on design and content management for static websites, and not on the development of interactive web applications.

2.2.1.3 Codux: Code-First Visual Editing

Codux is a visual IDE that works directly against a React codebase rather than a proprietary runtime. Key characteristics:

- **Code generation/sync**: Visual edits generate React components and styles in the repo (no separate model runtime)
- **Lower lock-in**: Output is plain code, maintainable outside the tool
- **Developer-centric**: Fits Git workflows and code review practices

This contrasts with model-at-runtime platforms such as Framer, and provides useful context for the trade-offs summarized by Sahay et al. **sahay2020supporting**.

2.2.1.4 Comparative Analysis

Feature	React	Codux	Framer	Webflow
State Management	`useState`/Fiber	`useState`/Fiber (code)	External JSON + runtime	No real state
Rendering	JSX/VDOM	JSX/VDOM (generates code)	`React.createElement`	Static DOM
Interactions	JS functions	JS functions (code-first)	Visual + code optional	Visual only
Serialization	Custom	Source code in repo	Built-in model store	HTML/CSS export
Dynamic Logic	Full JS support	Full JS support	Controlled JS sandbox	Limited animations

Table 2.1: Comparison of state and rendering approaches across tools, including a code-first option (Codux).

2.2.2 Low-Code Platform Reference Architecture

Low-code platforms share a recurring architecture that aligns closely with this thesis's system design. The study by Sahay et al. **sahay2020supporting** synthesizes common capabilities across platforms and proposes a conceptual architecture that provides useful context for understanding where visual builders fit within the broader ecosystem of development tools.

This subsection examines two complementary perspectives:

- a layered architecture that separates concerns across application composition, service and data integration, and deployment
- a component-based architecture that details the specific services and modules enabling low-code functionality. Taken together, these reference models position the proposed multi-layered visual builder within established patterns while highlighting its unique contributions.

Sahay et al. also propose a set of evaluation dimensions—such as user-interface tooling, interoperability, security, collaboration, reusability, scalability, business-logic tooling, build approach (code generation vs. model-at-runtime), deployment options, and target solution type—which expose trade-offs between accessibility and framework fidelity. These categories provide a backdrop for situating our system, particularly with respect to its treatment of the build process.

Figure **??** illustrates the layered architecture: application composition at the top, service and data integration in the middle, and deployment at the base. Within this framework, our visual editor is situated primarily in the application layer (toolbox, canvas, properties panel, layers view), while integration and deployment remain outside its scope. One clarification is worth noting: in the context of a visual builder, deployment involves two distinct concerns—the deployment of the platform itself and the runtime deployment of user-generated applications (for example, publishing projects to custom domains). Although this thesis does not address deployment in detail, making this distinction highlights that platform hosting and application publishing represent separate architectural challenges, each warranting dedicated research and analysis. The focus here remains on the architectural problem of achieving real-time editor reactivity and runtime component generation within the application layer.

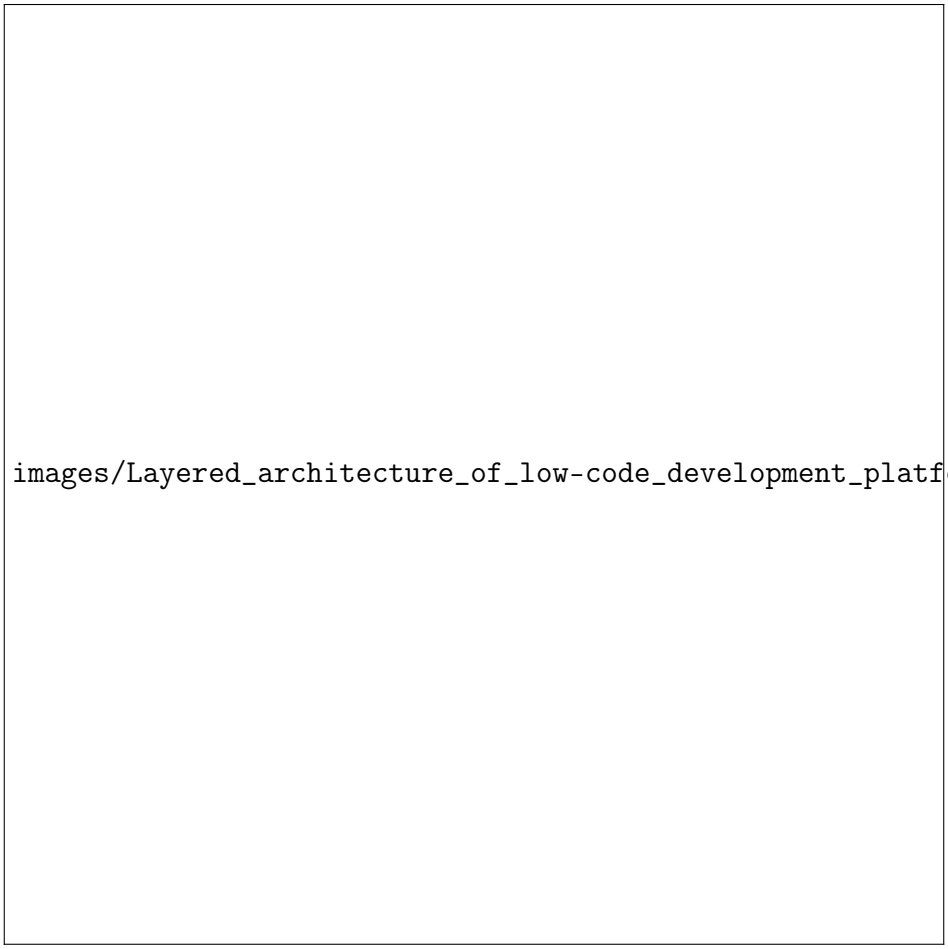

Figure 2.1: Layered architecture of low-code development platforms (adapted from Sahay et al. **sahay2020supporting**).

The component-based reference architecture in Figure **??** illustrates the sequential pipeline characteristic of most LCDPs. In this model, a graphical environment produces an abstract representation (often in XML), which is passed through a code generator to produce source code, then compiled before it can be deployed. While this clear separation of modeling, generation, and execution supports abstraction and portability, it sacrifices immediacy: modifications in the visual editor only become visible after the full translation and compilation process is complete.

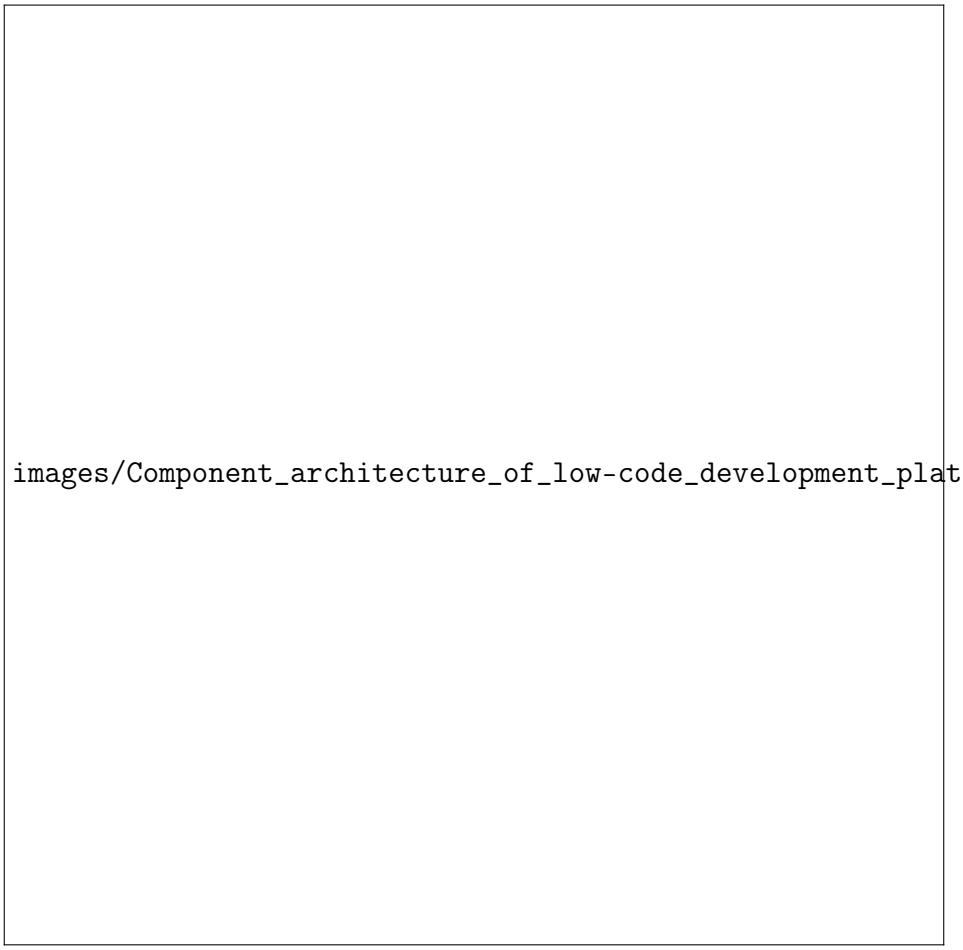

Figure 2.2: Component-based architecture of (adapted from Sahay et al. **sahay2020supporting**).

By contrast, the React-based visual builder developed in this thesis collapses this pipeline into a continuous reactive loop. Instead of XML, it employs a unified, serializable MobX State Tree component model. React itself functions as a real-time compiler: each mutation to the component tree is recursively translated through createElement into a virtual DOM, reconciled against the previous tree, and rendered in place. Visual edits made on the canvas, in the properties panel, or in the layers panel are instantly reflected as live, interactive previews. In this way, modeling and execution are unified within the same loop, combining the immediacy of a design tool with the full expressiveness of the underlying framework.

2.3 Relevant Technological Concepts

This section establishes the technological foundations necessary for understanding the multi-layered visual application builder. Rather than providing exhaustive technical documentation, the focus is on core concepts that enable runtime component generation, reactive state management, and efficient visual editing interactions.

The technological landscape for visual builders spans several domains: React's component model and rendering architecture provide the foundation for dynamic UI generation; reactive programming paradigms enable efficient state synchronization across editor layers; and modern JavaScript meta-programming techniques allow treating component configurations as manipulable data structures. Understanding these concepts is essential for grasping how the proposed system bridges visual editing interfaces with functional React applications.

The following subsections introduce these concepts at a level accessible to readers with basic React and JavaScript knowledge, emphasizing their relevance to visual editor architecture rather than implementation specifics, which are detailed in Chapters 5 and 6.

2.3.1 JavaScript DOM Manipulation

The Document Object Model (DOM) emerged as a critical bridge between static HTML content and dynamic, interactive capabilities in web browsers. In the early web, pages were static renderings with no mechanism for post-load manipulation. With the advent of JavaScript in 1995, browser vendors introduced ad-hoc, proprietary APIs, such as document.layers in Netscape Navigator and document.all in Internet Explorer, to allow scripts to access and alter page content **holzner2002insidejavascript**. These inconsistencies led to fragmentation and significant cross-browser compatibility issues. To resolve this, the W3C published the DOM Level 1 Recommendation in 1998, establishing a standardized interface for navigating and manipulating document structure and content **w3c1998dom**. Subsequent iterations DOM Level 2 (2000) and DOM Level 3 (2004) expanded the model with event handling, CSS integration, and XML support, paving the way for richer web interactions **w3c2000dom2**, **w3c2004dom3**.

The establishment of the DOM enabled a new paradigm of interactive web development. It allowed JavaScript to programmatically modify content, respond to user events, and adjust presentation dynamically, all without requiring a full page reload. This capability, in conjunction with the introduction of the XMLHttpRequest API in the early 2000s, made the Ajax model possible, which in turn fueled the rise of so-called "Web 2.0" applications with real-time, asynchronous updates **garrett2005ajax**. The DOM's live, mutable tree structure thus became the foundation for modern reactive frameworks such as React, Angular, and Vue. In particular, React builds upon the DOM by introducing the virtual DOM, an in-memory representation of the UI that is efficiently reconciled with the real DOM, ensuring that only the necessary changes are applied to the interface. Without the DOM as both a programmable abstraction and rendering target, neither Ajax-driven interactivity nor the component-based architectures of today's frameworks would be feasible.

2.3.2 React and its rendering pipeline

React is a JavaScript library for building user interfaces that introduced a declarative programming model, where interfaces are described as trees of components rather than through imperative DOM manipulation. Components can either be primitive host elements such as a `div`, or user-defined functions and classes that themselves compose other components. At the heart of React's architecture lies the idea of maintaining a virtual representation of this component tree and synchronizing it efficiently with the Document Object Model (DOM) whenever the underlying state or props change **chavan2021jsx**, **reactdocs_virtualdom**. This abstraction makes React a compelling foundation for visual application builders, since it allows changes in configuration to be reflected as live, interactive updates without the need for manual DOM handling.

2.3.2.1 JSX Compilation Pipeline

One of React's central innovations is JSX (JavaScript XML), a syntax extension that allows developers to write component structures in a markup-like form directly within JavaScript. Although JSX resembles HTML, it is not understood by browsers. Instead, it must be compiled, commonly via , into plain JavaScript. In earlier React versions, each JSX element was transformed into a call to `React.createElement`, returning an immutable *ReactElement* object. These ReactElements encode the type, properties, and children of a node, forming the blueprint for React's rendering process. **reactdocs_jsx**, **react17_jsx_runtime**

```
// JSX syntax
<button onClick={() => setCount(count + 1)}>
  {count}
</button>

// Compiled form (React <17)
React.createElement("button",
  { onClick: () => setCount(count + 1) },
  count
);
```

Listing 2.1: Legacy JSX to createElement Transformation

With the introduction of React 17, a new *JSX Transform* was released. Instead of compiling JSX into `React.createElement`, the compiler now emits calls to lightweight helper functions `jsx` and `jsxs` provided by the `react/jsx-runtime` entry point **react17_jsx_runtime**. This removes the requirement to import React explicitly in every file and enables further optimizations in bundle size and tooling support.

```
// JSX syntax
<button onClick={() => setCount(count + 1)}>
  {count}
</button>

// Compiled form (React 17+)
import { jsx as _jsx } from "react/jsx-runtime";

_jsx("button", { onClick: () => setCount(count + 1), children: count });
```

Listing 2.2: New JSX Runtime Transformation (React 17+)

By treating JSX as declarative sugar for producing ReactElements, React decouples user interface specification from low-level DOM operations. The shift from `createElement` to the new runtime-based transform reflects React's long-term strategy of simplifying developer experience and enabling runtime systems, such as visual builders, to generate components as structured data rather than through direct DOM manipulation.

2.3.2.2 Virtual DOM and Reconciliation

The React runtime operates by maintaining a Virtual DOM: an in-memory tree of ReactElements that mirrors the intended structure of the interface. When state or props change, React regenerates the relevant parts of this virtual tree and compares it against the previous version. Through its reconciliation process, React applies only the minimal set of mutations needed to update the real DOM, using heuristics that optimize for performance in common scenarios such as list reordering **reactdocs_virtualdom**, **geeksforgeeks_reconciliation**, **guha2010programming**.

For visual editors, this model is especially advantageous: interactions like moving a component, toggling a property, or reorganizing a hierarchy can be reflected immediately without re-rendering the entire interface. This efficiency is critical for delivering a responsive editing experience.

2.3.2.3 React Fiber and Concurrent Rendering

With React 16, the reconciliation engine was fundamentally redesigned under the name Fiber, introducing an asynchronous model of rendering. A fiber is a lightweight object representing a unit of work in the rendering pipeline. The system organizes these fibers into a "work-in-progress" tree that React processes incrementally before committing the final result to the DOM.

Rendering is divided into two phases: an interruptible render phase (building the work-in-progress tree using `beginWork()` and `completeWork()`), and a synchronous commit phase (`commitWork()`), which applies mutations to the DOM. This split ensures that while updates can be scheduled flexibly, DOM consistency is always preserved **react_fiber_explained**, **react_fiber_architecture**.

Concurrent features enabled by Fiber, such as interruptible rendering, priority-based scheduling,

and time-slicing, allow React to keep applications interactive even during large updates. For visual application builders, these capabilities are essential: dragging a component across a canvas or typing into a property panel can be prioritized over background re-renders, ensuring fluid responsiveness **react_fiber_concurrent**, **cooper2021fiber**.

Taken together, JSX, the Virtual DOM, and Fiber's concurrent reconciliation establish React as a runtime that behaves like a real-time compiler of user interfaces. Each change to a component's configuration is translated into a ReactElement tree, reconciled against the prior version, and rendered to the DOM with minimal overhead—providing live previews that feel immediate while retaining the power of a full programming model.

2.3.3 Next.js Framework

While React is a front-end library focused purely on rendering user interfaces in the browser, it does not provide features for routing, data fetching, or server-side logic. By contrast, Next.js is a full-stack framework built on top of React that combines a powerful front-end component model with a server runtime for handling backend concerns **vercel_nextjs**.

This distinction is essential: Next.js applications can define *API routes* and server-side logic that run securely on the server, never exposed to the client. This includes tasks such as database queries, authentication flows, or calls to external APIs. The separation of concerns ensures that sensitive operations remain on the server, while the client receives only the data needed for rendering.

For the purposes of this project, Next.js was chosen over plain React for several reasons:

- **Full-stack capabilities**: API routes and server functions enable integrating databases and external services without exposing business logic to the client.
- **Hybrid rendering**: Components can be rendered server-side, statically, or client-side depending on context, improving performance and SEO.
- **File-based routing**: Automatic routing simplifies the creation of multi-page editor interfaces.
- **Production-ready defaults**: Features such as code splitting, image optimization, and deployment pipelines reduce boilerplate and accelerate development.

In the context of a visual application builder, Next.js ensures that the editor can operate as a cohesive full-stack system: the user interface is rendered with React, while server-side logic (such as project persistence, authentication, and external API calls) is securely handled on the backend. This combination of declarative UI and server-side infrastructure makes Next.js the pragmatic choice for this thesis over a plain React setup.

2.3.4 MobX and MobX State Tree Foundations

To understand the state management backbone of our visual builder prototype, it is essential to explore how MobX and MobX State Tree (MST) enable reactive, structured, and serializable state handling.

2.3.4.1 Reactive State with MobX

MobX is a reactive state management library that implements the paradigm of *Transparent Functional Reactive Programming (TFRP)* **mobx_docs**. In this paradigm:

- **Functional** – Derived values (e.g., computed properties or React component renders) are expressed as pure functions of observable state: given the same state, they always yield the same result.
- **Reactive** – When observables change, all dependent values and observers automatically update.
- **Transparent** – Dependency tracking is inferred automatically at runtime without the need for manual subscription logic.

Comparison with React's Pull-Based Model. React employs a *pull-based* model: when state changes, components re-render, pulling in fresh values and recomputing derived results. MobX, by contrast, implements a *push-based* model: observables notify only the computations and components that actually depend on them.

```
function Cart() {
  const [items, setItems] = useState([]);

  // Derived value recalculated on every render
  const totalPrice = items.reduce((sum, item) => sum + item.price, 0);

  return (
    <div>
      <h3>Total: {totalPrice}</h3>
      <ul>{items.map(item => <li key={item.id}>{item.name}</li>)}</ul>
    </div>
  );
}
```

Listing 2.3: React Pull-Based State Derivation

```
class Cart {
  items = [];
  constructor() { makeAutoObservable(this); }

  get totalPrice() {
    return this.items.reduce((sum, item) => sum + item.price, 0);
  }
}

const CartView = observer(({ cart }) => (
  <div>
    <h3>Total: {cart.totalPrice}</h3>
    <ul>{cart.items.map(item => <li key={item.id}>{item.name}</li>)}</ul>
  </div>
));
```

Listing 2.4: MobX Push-Based Transparent Reactivity

In React, `totalPrice` is recalculated on every render (pull). In MobX, it is cached and only recomputed when its dependencies change, and any observing component is automatically updated (push).

2.3.4.2 What MST Adds to MobX

MobX State Tree (MST) builds on MobX by introducing structured, typed state trees with built-in serialization capabilities. Key architectural features include:

- **Living state trees**: MST represents application state as a live, mutable tree, enriched by runtime type information and strict update constraints **mobx_docs**.
- **Snapshots and patching**: The tree can be serialized into immutable snapshots, enabling history tracking, time travel, and easy persistence **mobx_docs**.

- **Typed models and actions**: Each node in the tree is defined via a model with its own actions—explicit methods to mutate state—ensuring traceability and type safety.
- **Reconciliation**: Applying a snapshot to an existing tree reuses existing nodes where possible, mimicking React's diffing philosophy at the data layer.

2.3.4.3 Decoupling Domain and UI

One of MST's main advantages is the decoupling of state and business logic from the user interface. Rather than co-locating state within React components (via hooks), MST encourages modeling the domain problem first: users, projects, or UI elements become structured models within the state tree. The UI layer then acts as a projection of this domain model. As Weststrate, the creator of MobX, argues, this makes it possible to treat the *UI as an afterthought*, allowing applications to evolve around a well-modeled domain state **weststrate_ui**.

Illustrative Tabular Comparison

	React (JSX UI)	**MobX State Tree (MST)**
State	immutably updated (via hooks)	mutable but type-checked
Reactivity Model	pull (component-triggered re-render)	push (observable-triggered)
Data Structure	UI tree built on render	Living state tree models
Serialization	manual (custom logic)	automatic snapshots / patches
Updates	declarative UI changes	in-place actions with history support

Table 2.2: React vs. MST: contrasting reactivity and data handling models

2.3.4.4 Relevance of MST in the context of Visual Application Builders

MobX State Tree provides an elegant and performant state management foundation for the visual builder prototype. Its structured, observable state tree aligns perfectly with the editor's requirements: serializable component models, responsive updates, history tracking, and seamless integration with React's JSX runtime make it a compelling choice for building interactive, domain-driven user interfaces.

3. Methodology

The methodology of this thesis reflects the path from an exploratory investigation to a structured validation of a prototype for a multi-layered application builder. Rather than following a strictly linear plan from the beginning, the project evolved through distinct stages: an initial phase of open experimentation, an analytical phase of reverse-engineering existing solutions, and a subsequent shift into a more systematic, iterative prototyping approach. This chapter outlines that process and explains how each phase contributed to the outcome of the project.

3.1 Research Approach

The overall research approach can be described as iterative prototyping supported by exploratory research methods. The project began without a rigid development model, instead relying on rapid experimentation, trial-and-error coding, and the use of AI tools for technology scouting and feasibility assessments. Over time, this exploratory process gave way to a more structured methodology, in which requirements were formalized, architectural decisions were documented, and systematic validation techniques were applied. This combination of unstructured exploration and structured iteration made it possible both to survey a broad solution space and to deliver a coherent, validated prototype.

3.2 Exploratory Prototyping and AI-Assisted Research

In the earliest stage, the development was highly experimental. Over a period of several months, a prototype was created through what could be described as "", an exploratory style of programming where ideas were tested directly in code without prior formalization. This was supported by extensive use of AI systems to research frameworks, libraries, and design patterns that might be suitable for building an application builder. While unstructured, this phase proved essential for quickly identifying technological opportunities and constraints. It allowed the project to answer early feasibility questions, such as whether Next.js could serve as the backbone of a multi-layered system, and which supporting tools might be required for database management, state handling, and deployment.

3.3 Reverse Engineering of Existing Solutions

Following the exploratory phase, attention turned to existing application builders and low-code platforms. These systems were analyzed and, where possible, partially reverse-engineered in order to uncover how they structured their architectures, implemented modularity, and balanced functional against non-functional requirements. This comparative analysis served two purposes:

first, it contextualized the prototype within a wider ecosystem of solutions, and second, it provided concrete reference points against which design decisions could be validated. The insights gained here fed directly into the later architectural modeling of the prototype.

3.4 Transition to Structured Iterative Development

Once the technological landscape had been explored and comparative insights established, the project shifted into a more systematic methodology. Requirements identified in Chapter 4 were translated into concrete architectural and implementation decisions. The prototype was developed in iterative cycles, where each iteration extended functionality, addressed non-functional requirements such as modularity and performance, and incorporated feedback from prior evaluations. Tools such as version control and containerization were used to ensure reproducibility and maintainability. Coding conventions and automated checks supported consistency and quality during development.

3.5 Validation Strategy

Validation was conducted continuously throughout the iterative cycles. Functional validation involved both manual and automated testing, ensuring that each requirement defined in Chapter 4 was correctly implemented. Non-functional validation focused on measuring performance, maintainability, and scalability, which are particularly important for a multi-layered builder architecture. In addition, the prototype was compared against the features and limitations of existing solutions, which highlighted both its strengths and areas where further work would be needed. Documenting limitations and unresolved challenges was treated as an integral part of the validation process, laying the groundwork for the evaluation in Chapter 6.

4. Requirements Analysis

This chapter establishes the functional and non-functional requirements for the research prototype, including scope, modularization, integration, result artifacts, and data protection. The requirements for the visual application builder are derived from the core user needs and system constraints identified in previous chapters.

4.1 Functional Requirements

The functional requirements are defined in table **??**. They are organized here as mandatory and optional, and presented in a structured table for clarity and traceability. The classification of requirements as MUST reflects their necessity in a system that serves as a proof of concept. For the research prototype, selected MUST requirements were only partially implemented to validate feasibility under time constraints. The exact scope of the requirements is defined in section **??**.

ID	Requirement	Type/Priority
FR-1	Immediate updates across canvas, layers panel, and properties panel	Core Editing / MUST
FR-2	Real-time rendering of recursive MST component tree with responsive breakpoints	Rendering / MUST
FR-3	Visual selection of components on the canvas by clicking on them	Selection / MUST
FR-4	Hierarchical navigation and drag-and-drop reordering in a tree view	Structure Management / MUST
FR-5	Context-sensitive editing for props (style, content, attributes)	Properties Editing / MUST
FR-6	Serializable component model (id, type, responsive props, children)	Data Model / MUST
FR-7	Breakpoint-aware prop resolution (inheritance from primary, overrides per breakpoint)	Responsiveness / MUST
FR-8	Project persistence: create, read, update, delete; multiple projects per user	Persistence / CAN
FR-9	Execution of interactions (e.g., onClick) in preview mode	Interactivity / CAN
FR-10	Undo/Redo via snapshot-based history navigation	Usability / CAN

Table 4.1: Functional Requirements for the Visual Application Builder

4.2 Non-functional Requirements

Non-functional requirements (NFRs) specify the quality attributes and constraints of a system, rather than its functional behavior. As Sommerville notes, "non-functional requirements are requirements that are not directly concerned with the specific services delivered by the system to its users. They instead define constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc." **sommerville2016**.

ID	Requirement	Type/Priority
NFR-1	Low-latency user interactions (<100ms perceived delay for pan, zoom, selection)	Usability / MUST
NFR-2	GPU-accelerated rendering of canvas transforms (pan/zoom via CSS transforms)	Performance / MUST
NFR-3	Efficient handling of medium-sized component trees (hundreds of nodes) without frame drops	Performance / MUST
NFR-4	Shared transform state (pan, zoom) accessible across canvas and overlays without triggering React re-renders	Architecture / MUST
NFR-5	Codebase structured for incremental extension (tools, overlays, collaboration) without core rewrite	Maintainability / MUST
NFR-6	Error-tolerant persistence: data must remain consistent and recoverable after system failure	Reliability / MUST

Table 4.2: Non-Functional Requirements for the Visual Application Builder

4.3 Research Design and Scope

The research design of this thesis emphasizes feasibility validation over completeness. The prototype is intentionally restricted to a minimal set of core mechanisms that collectively demonstrate the viability of a multi-layered visual application builder. Rather than attempting to deliver a production-ready system, the implementation focuses on proving three central aspects: state synchronization across interface layers, runtime updates on an infinite canvas, and the modular interoperability of component-based structures.

Within this scope, the prototype integrates an infinite canvas with optimized pan and zoom, a MobX-State-Tree (MST) component tree with responsive properties, and a selection system that synchronizes the canvas with the layers panel. A lightweight styling system supports inline styles, attributes, and responsive overrides. These elements form the foundation for a responsive and interactive editing environment, sufficient to validate the central research questions.

4.4 Result Artifacts

The outcomes of the prototype are twofold. First, as **primary artifacts**, the system delivers functional modules: the infinite canvas, recursive renderer, selection overlay, layers panel, and properties panel. Together, these modules demonstrate real-time reactivity across interface layers and the ability to render from a serializable MST component tree at runtime.

Second, as **conceptual contributions**, the research provides architectural patterns for modular visual builders, validates breakpoint inheritance and responsive resolution within MST, and offers recommendations for extensibility, such as pluggable components and future collaboration hooks. These artifacts are less about software maturity than about informing future iterations of visual builder architectures.

4.5 Data Protection Aspects

The research prototype processes only configuration data and does not involve personal information, thereby presenting minimal data protection risks under GDPR. Nevertheless, for future production-grade deployment, it is crucial to adopt established privacy frameworks such as *Privacy by Design* and *Data Minimization*. The principle of Data Minimization—collecting, retaining, and processing only the data strictly necessary for a specific purpose—is a cornerstone of effective data protection **ganesh2024dataMinimization**. This aligns with broader Privacy by Design approaches, which emphasize embedding privacy considerations throughout the system lifecycle **jezova2020privacyByDesign**, **danezis2015privacyByDesign**.

In the context of a multi-user or multi-project architecture, ensuring tenant isolation is essential to maintain data confidentiality and separation. Numerous studies recommend robust multi-tenancy isolation via architectural segregation—ranging from isolated schemas or databases to access control mechanisms grounded in RBAC or ABAC models **chippagiri2025multiTenantSecurity**, **das2019multiTenan**
Additionally, prudent secret management and authentication mechanisms—such as rotating environment-level credentials and applying fine-grained access control—are foundational to system resilience.

While these privacy safeguards are outside the immediate scope of the feasibility prototype, they remain vital for future development cycles. Acknowledging and planning for these measures, proactively positions the architecture to meet regulatory expectations and build user trust in subsequent, production-focused iterations.

5. Conception & Design

The conception and design of the multi-layered visual application builder emerged from a systematic exploration of the fundamental challenges facing modern web development tools. This chapter documents the design philosophy, architectural principles, and strategic decisions that shaped the development of a system capable of bridging the gap between visual editing and runtime functionality.

The core challenge addressed by this research lies in the inherent tension between two competing requirements: the need for immediate visual feedback during the editing process, and the preservation of native React component behavior in the final application. Traditional approaches to visual editors often sacrifice one for the other, leading to either poor editing experiences or compromised runtime performance. The design presented in this chapter demonstrates how these seemingly contradictory requirements can be reconciled through careful architectural planning and innovative performance optimization techniques.

5.1 Design Philosophy and Approach

The design philosophy underlying this visual application builder is grounded in the principle that components should exist as unified, serializable data structures that can be seamlessly transformed between their editing and runtime representations. This approach challenges the conventional wisdom that visual editors require separate data models for design-time and runtime concerns.

The development process followed an iterative methodology that combined exploratory prototyping with systematic performance analysis. Initial prototypes focused on understanding the fundamental constraints of visual editors, particularly the performance bottlenecks that emerge when attempting to provide real-time visual feedback for complex component hierarchies. Through careful analysis of existing solutions, particularly Framer's architecture, key insights emerged about the importance of separating high-frequency operations from React's reconciliation process.

The design validation strategy emphasized measurable performance criteria alongside qualitative assessments of user experience and code maintainability. Each architectural decision was evaluated through multiple lenses: its impact on rendering performance, its contribution to system extensibility, and its alignment with established React patterns and best practices.

5.2 Core Architectural Principles

The architecture is founded upon four fundamental principles that address the primary challenges identified in the requirements analysis. These principles work together to create a cohesive system that maintains both editor responsiveness and runtime component integrity.

5.2.1 Unified Component Model

The first principle establishes that all elements within the system, whether they exist as part of hierarchical component trees or as floating canvas elements, should be represented using a single, consistent data structure. This unified component model eliminates the impedance mismatch that typically exists between editor representations and runtime components.

The component model extends beyond traditional React component definitions to include canvas-specific properties such as positioning coordinates, scale, rotation, and visibility constraints. These properties enable the system to support both nested component hierarchies (typical of React applications) and absolute-positioned floating elements (common in design tools) within the same architectural framework.

```
interface ComponentModel {
  id: string;
  type: string; // 'div', 'button', custom components
  props: Record<string, any>;
  children: ComponentModel[];

  // Canvas-specific properties
  canvasX?: number;
  canvasY?: number;
  canvasVisible: boolean;
}
```

Listing 5.1: Conceptual Component Model Structure

This unified approach provides several key benefits: consistent serialization mechanisms for all element types, simplified rendering logic that can handle both hierarchical and floating elements, and a foundation for responsive design through breakpoint-aware property resolution.

5.2.2 Performance-First Canvas Architecture

The second principle prioritizes performance optimization for canvas operations, recognizing that visual editors must maintain smooth 60fps interactions even with hundreds or thousands of elements. Traditional approaches that update individual element positions during pan and zoom operations quickly become performance bottlenecks as the number of elements grows.

The solution lies in adopting coordinate transformation rather than individual element updates. Instead of modifying the position of each element during canvas navigation, the system applies a single transform to a container element, effectively moving the entire viewport. This approach reduces the computational complexity of canvas operations from O(n) to O(1), where n represents the number of elements on the canvas.

The performance benefits extend beyond basic pan and zoom operations to include selection overlay updates, element highlighting, and other visual feedback mechanisms that must remain respon-

sive during user interactions.

5.2.3 Meta-Programming Component Generation

The third principle treats component configurations as serializable data that can be transformed into executable React components at runtime. This meta-programming approach enables visual manipulation of component properties while preserving full React functionality in the resulting applications.

The system maintains component definitions as structured data rather than as traditional React component code. During rendering, these data structures are dynamically transformed into React elements using React.createElement, allowing for real-time modification of component properties, hierarchy, and behavior without requiring code generation or compilation steps.

This approach enables several advanced features: responsive property resolution based on breakpoint contexts, dynamic component type switching, and real-time preview of component behavior changes. The meta-programming foundation also facilitates future extensibility through plugin systems and custom component registration.

5.2.4 Reactive State Management

The fourth principle establishes a reactive state management system that can efficiently handle both editor-specific updates (selection changes, property modifications) and runtime component behavior (user interactions, state changes). The system must maintain consistency between these two reactive contexts while avoiding performance degradation.

The solution employs MobX State Tree as the foundation for reactive state management, providing automatic dependency tracking and efficient updates. However, performance-critical operations such as canvas transforms are handled through direct DOM manipulation to bypass React's reconciliation process entirely.

This hybrid approach ensures that editor operations remain responsive while preserving the predictable state management patterns that React developers expect. The reactive system automatically propagates changes from the central state tree to all observing components, ensuring that the user interface remains synchronized with the underlying data model.

5.3 System Architecture Overview

The system architecture organizes functionality across four distinct layers, each serving a specific purpose while maintaining clear separation of concerns and efficient data flow patterns. This layered approach enables independent optimization of each layer while ensuring seamless integration

across the entire system.

The infinite canvas layer provides the primary visual editing surface, implementing high-performance pan and zoom through GPU-accelerated CSS transforms. This layer manages the coordinate systems for both hierarchical components and floating elements, handling the complex transformations required to maintain accurate positioning during canvas navigation.

The component management layer offers tools for creating, organizing, and navigating component hierarchies. This includes a drag-and-drop component palette for adding new elements, a hierarchical layers panel for navigation and organization, and selection mechanisms that work across both nested and floating elements.

The properties panel layer provides context-sensitive editing interfaces that adapt based on the currently selected component type. This layer implements responsive property editing, allowing users to define different values for the same property across multiple breakpoints, with sophisticated inheritance and cascading behavior.

The overlay system layer handles visual feedback for selected elements, implementing coordinate transformation between canvas space and screen space to ensure accurate overlay positioning regardless of canvas zoom and pan state. This layer demonstrates the performance benefits of the subscription-based update pattern, avoiding React re-renders for high-frequency updates.

5.4 Responsive Design Strategy

The responsive design strategy addresses the challenge of creating applications that adapt to different screen sizes and device capabilities while maintaining visual consistency and functional integrity. The system implements a mobile-first approach with sophisticated property inheritance and cascading behavior.

Component properties can be defined as responsive maps, where different values are specified for different breakpoints. For example, a component's width property might be defined as follows:

```
const rootComponent = createComponent('div', {
  style: {
    width: {
      desktop: '1280px',
      tablet: '768px',
      mobile: '320px'
    },
    backgroundColor: { desktop: '#4A90E2' }
  }
});
```

Listing 5.2: Responsive Property Example

The system automatically resolves the appropriate value based on the current breakpoint context, following a predictable inheritance chain that cascades from smaller to larger breakpoints when specific values are not defined. This approach ensures that components behave predictably across different screen sizes while minimizing the amount of configuration required from users.

The responsive resolution algorithm prioritizes direct breakpoint matches, falls back to primary breakpoint values, and then cascades through the breakpoint hierarchy to find appropriate values. This approach ensures that components behave predictably across different screen sizes while minimizing the amount of configuration required from users.

The responsive system extends beyond simple property values to include component visibility constraints, allowing elements to be shown or hidden based on breakpoint ranges. This capability enables sophisticated responsive layouts that can restructure themselves based on available screen space.

5.5 Performance Optimization Strategy

The performance optimization strategy recognizes that visual editors have fundamentally different performance characteristics than traditional web applications. While most web applications prioritize initial load time and perceived performance, visual editors must maintain consistent 60fps interactions during intensive operations such as canvas navigation and element manipulation.

The optimization strategy separates high-frequency operations from React's reconciliation process through direct DOM manipulation and subscription-based updates. Canvas transforms are applied directly to DOM elements using CSS transforms, bypassing the virtual DOM entirely for these performance-critical operations.

Selection overlays and visual feedback elements use a subscription pattern that receives updates through callback functions rather than React state changes. This approach eliminates unnecessary re-renders while maintaining accurate synchronization with canvas state.

GPU optimization techniques ensure that transform operations are handled by the graphics hard-

ware rather than the CPU. Strategic use of CSS properties such as will-change, transform-origin, and compositing hints instructs the browser to create optimized rendering layers for smooth animations and interactions.

The performance strategy also addresses memory management through efficient subscription cleanup, cached DOM queries, and strategic use of React.memo and useCallback to prevent unnecessary component re-renders in the user interface layers.

This comprehensive approach to performance optimization enables the system to maintain professional-grade responsiveness even with complex component hierarchies and intensive user interactions, demonstrating that web-based visual editors can achieve performance characteristics comparable to native desktop applications.

This design foundation establishes the conceptual framework for a visual editor that can compete with professional design tools while maintaining the flexibility and extensibility that modern web development demands. The principles outlined in this chapter guide the implementation decisions documented in the following chapter, ensuring that the technical implementation aligns with the strategic design goals.

5.6 Future Considerations

While the current prototype demonstrates the feasibility of the core architectural principles, several areas present opportunities for future development and research. The dual-reactivity challenge, where editor reactivity must coexist with preserved runtime component interactivity, represents a complex architectural problem that extends beyond the scope of this initial implementation.

The current system focuses primarily on the editor reactivity aspects, ensuring smooth visual editing experiences through reactive state management and performance optimization. The runtime component interactivity aspect, while conceptually addressed through the serializable component model, would require additional research and development to fully realize in production applications.

Future work might explore advanced state synchronization mechanisms that can maintain component state across editing and runtime contexts, sophisticated component lifecycle management that preserves user interactions during live editing sessions, and integration patterns that enable seamless transitions between editing and preview modes without losing application state.

The extensibility architecture also presents opportunities for community-driven development, where third-party developers could contribute custom components, editing tools, and export formats through well-defined plugin interfaces. This ecosystem approach could significantly expand the capabilities of the visual editor while maintaining the performance and architectural integrity established by the core system.

6. Implementation

This chapter documents the technical implementation of the multi-layered visual application builder, translating the architectural principles established in Chapter 5 into working code. The implementation demonstrates how modern web technologies can be orchestrated to create a professional-grade visual editor that maintains both editing responsiveness and runtime component integrity.

The implementation strategy prioritizes incremental development and systematic validation of each architectural component. Rather than attempting to build the complete system simultaneously, the development process focused on establishing robust foundations for state management, canvas performance, and component rendering before integrating the complete user interface layers.

The technical choices documented in this chapter reflect the balance between innovation and pragmatism required for a research prototype. While exploring novel approaches to performance optimization and state management, the implementation maintains compatibility with established React patterns and modern web development practices to ensure maintainability and extensibility.

6.1 State Management Foundation

The implementation of the state management layer establishes the reactive foundation that enables real-time visual editing while maintaining data integrity and type safety. MobX State Tree was selected as the primary state management solution due to its unique combination of automatic reactivity, runtime type validation, and snapshot-based serialization capabilities.

The state management architecture addresses several critical requirements simultaneously: the need for fine-grained reactivity that can efficiently update user interface elements when component properties change, the requirement for complete state serialization to support project persistence and collaboration features, and the necessity of maintaining type safety across complex data transformations.

6.1.1 Unified Component Model

The core component model represents the most critical architectural decision in the entire system, as it must serve both as a data structure for visual editing and as a blueprint for runtime React component generation. The implementation adopts Framer's "everything is a component" philosophy, eliminating artificial distinctions between different types of canvas elements.

```typescript
// src/models/ComponentModel.ts
import { types, Instance, SnapshotIn, getParent } from 'mobx-state-tree';

// Supported component types
enum ComponentTypeEnum {
  HOST = 'HOST',      // DOM elements: div, button, img, etc.
  FUNCTION = 'FUNCTION' // Custom React components
}

// Base component model without recursive children
const ComponentBase = types.model('ComponentBase', {
  // Core identity and type information
  id: types.identifier,
  type: types.string,
  componentType: types.enumeration(Object.values(ComponentTypeEnum)),
  props: types.optional(types.frozen<PropsRecord>(), {}),

  // Canvas positioning for root-level components (Framer-style)
  canvasX: types.maybe(types.number),
  canvasY: types.maybe(types.number),
  canvasScale: types.optional(types.number, 1),
  canvasRotation: types.optional(types.number, 0),
  canvasZIndex: types.optional(types.number, 0),

  // Responsive visibility constraints
  visibleFromBreakpoint: types.maybe(types.string),
  visibleUntilBreakpoint: types.maybe(types.string),

  // Canvas-level properties
  canvasVisible: types.optional(types.boolean, true),
  canvasLocked: types.optional(types.boolean, false),
});

// Final component model with recursive children and actions
const ComponentModel = ComponentBase
  .props({
    children: types.optional(types.array(types.late(() => ComponentModel))
      , [])
  })
  .actions(self => ({
    // Canvas positioning actions
    updateCanvasTransform(updates: {
      x?: number; y?: number; scale?: number;
      rotation?: number; zIndex?: number;
    }) {
      if (updates.x !== undefined) self.canvasX = updates.x;
      if (updates.y !== undefined) self.canvasY = updates.y;
      if (updates.scale !== undefined)
        self.canvasScale = Math.max(0.1, Math.min(10, updates.scale));
      if (updates.rotation !== undefined)
        self.canvasRotation = updates.rotation % 360;
      if (updates.zIndex !== undefined)
        self.canvasZIndex = updates.zIndex;
```

```
    },

    // Breakpoint visibility management
    setBreakpointVisibility(fromBreakpoint?: string, untilBreakpoint?:
        string) {
      self.visibleFromBreakpoint = fromBreakpoint;
      self.visibleUntilBreakpoint = untilBreakpoint;
    },

    // Canvas state management
    toggleCanvasVisibility() {
      self.canvasVisible = !self.canvasVisible;
    },

    toggleCanvasLock() {
      self.canvasLocked = !self.canvasLocked;
    }
  }))
  .views(self => ({
    // Component type checking
    get isHostElement(): boolean {
      return self.componentType === ComponentTypeEnum.HOST;
    },

    get isRootCanvasComponent(): boolean {
      return self.canvasX !== undefined && self.canvasY !== undefined;
    },

    // CSS transform generation for canvas positioning
    get canvasTransform(): string {
      if (!this.isRootCanvasComponent) return '';
      return `translate(${self.canvasX}px, ${self.canvasY}px) scale(${self
          .canvasScale}) rotate(${self.canvasRotation}deg)`;
    },

    // Bounding box calculation for overlays
    get canvasBounds() {
      if (!this.isRootCanvasComponent) return null;
      const width = self.props?.width || 200;
      const height = self.props?.height || 100;
      return {
        x: self.canvasX!,
        y: self.canvasY!,
        width: typeof width === 'string' ? parseInt(width) : width,
        height: typeof height === 'string' ? parseInt(height) : height,
      };
  }
}));

export default ComponentModel;
export type ComponentInstance = Instance<typeof ComponentModel>;
export type ComponentSnapshotIn = SnapshotIn<typeof ComponentModel>;
```

Listing 6.1: ComponentModel MST Implementation

6.1.2 Responsive Property Resolution

The responsive property system represents one of the most sophisticated aspects of the implementation, enabling components to adapt their behavior and appearance across different screen sizes while maintaining intuitive inheritance patterns. The system supports complex responsive configurations where properties can be defined with different values for each breakpoint.

A practical example of responsive properties can be seen in the root component creation within the project initialization:

```
// src/stores/ProjectStore.ts (lines 71-77)
const rootComponent = createIntrinsicComponent('root-' + rootComponentId,
    'div', {
  style: {
    position: { [desktopBreakpointId]: '' },
    width: {
      [desktopBreakpointId]: '1280px',
      [tabletBreakpointId]: '768px',
      [mobileBreakpointId]: '320px'
    },
    height: { [desktopBreakpointId]: '1000px' },
    backgroundColor: { [desktopBreakpointId]: '#4A90E2' },
  },
});
```

Listing 6.2: Responsive Properties Example from ProjectStore

This example demonstrates how a single component can specify different width values for desktop, tablet, and mobile breakpoints, while the backgroundColor property is only defined for the desktop breakpoint, relying on the inheritance system for other screen sizes.

The responsive resolution algorithm implements a sophisticated cascading system that prioritizes direct matches while providing predictable fallback behavior:

```typescript
// src/models/ComponentModel.ts (continued)

// Helper function to resolve responsive values with inheritance
function resolveResponsiveValue(
  map: Record<string, any>,
  breakpointId: string,
  ordered: { id: string; minWidth: number }[],
  primaryId: string
) {
  // 1. Direct breakpoint match - highest priority
  if (map[breakpointId] !== undefined) return map[breakpointId];

  // 2. Primary breakpoint fallback
  if (map[primaryId] !== undefined) return map[primaryId];

  const idx = ordered.findIndex(b => b.id === breakpointId);
  if (idx === -1) return map.base;

  // 3. Cascade to smaller breakpoints first (mobile-first approach)
  for (let i = idx - 1; i >= 0; i--) {
    const id = ordered[i].id;
    if (map[id] !== undefined) return map[id];
  }

  // 4. Cascade to larger breakpoints
  for (let i = idx + 1; i < ordered.length; i++) {
    const id = ordered[i].id;
    if (map[id] !== undefined) return map[id];
  }

  // 5. Final fallback to base value
  return map.base;
}

// Extended ComponentModel with responsive property resolution
const ComponentModelExtended = ComponentModel.views(self => ({
  // Resolve all properties for a specific breakpoint
  getResolvedProps(
    breakpointId: string,
    allBreakpoints: BreakpointType[],
    primaryId: string
  ) {
    const ordered = [...allBreakpoints].sort((a,b) => a.minWidth - b.
       minWidth);
    const bpIds = new Set(ordered.map(bp => bp.id));

    const attributes: Record<string, any> = {};
    const style: Record<string, any> = {};

    // Process all component properties
    for (const [key, raw] of Object.entries(self.props)) {
      if (key === 'style') continue; // Handle style separately

```

```
53        if (isResponsiveMap(raw, bpIds)) {
54          // Resolve responsive property
55          const val = resolveResponsiveValue(raw, breakpointId, ordered,
                primaryId);
56          if (CSS_PROP_SET.has(key)) {
57            style[key] = val;
58          } else {
59            attributes[key] = val;
60          }
61        } else {
62          // Static property
63          if (CSS_PROP_SET.has(key)) {
64            style[key] = raw;
65          } else {
66            attributes[key] = raw;
67          }
68        }
69      }
70
71      return { attributes, style };
72    }
73  }));
74
75  // Helper to detect responsive property maps
76  function isResponsiveMap(value: any, breakpointIds: Set<string>): boolean
        {
77    if (typeof value !== 'object' || value === null) return false;
78    const keys = Object.keys(value);
79    return keys.some(key => breakpointIds.has(key) || key === 'base');
80  }
```

Listing 6.3: Responsive Property Resolution Implementation

6.1.3 Project and Page Model Implementation

The hierarchical project structure organizes components and manages breakpoints:

```typescript
// src/models/ProjectModel.ts
import { types, Instance, SnapshotIn } from 'mobx-state-tree';
import PageModel from './PageModel';
import { BreakpointModel } from './BreakpointModel';

const ProjectMetadataModel = types.model('ProjectMetadata', {
  title: types.string,
  description: types.optional(types.string, ''),
  createdAt: types.optional(types.Date, () => new Date()),
  updatedAt: types.optional(types.Date, () => new Date()),
});

export const ProjectModel = types
  .model('Project', {
    id: types.identifier,
    metadata: ProjectMetadataModel,
    breakpoints: types.map(BreakpointModel),
    primaryBreakpoint: types.reference(BreakpointModel),
    pages: types.map(PageModel),
  })
  .actions(self => ({
    // Breakpoint management
    addBreakpoint(label: string, minWidth: number): BreakpointType {
      const breakpointId = uuidv4();
      self.breakpoints.set(breakpointId, {
        id: breakpointId, label, minWidth
      });
      return self.breakpoints.get(breakpointId)!;
    },

    updateBreakpoint(breakpointId: string, updates: Partial<
       BreakpointSnapshotIn>) {
      const breakpoint = self.breakpoints.get(breakpointId);
      if (breakpoint) {
        applySnapshot(breakpoint, { ...getSnapshot(breakpoint), ...updates
            });
        self.metadata.updatedAt = new Date();
      }
    },

    // Page management
    addPage(slug: string, title: string): PageModelType {
      const pageId = uuidv4();
      const page = PageModel.create({
        id: pageId,
        slug,
        metadata: { title },
        rootComponent: createSampleComponentTree(),
        rootCanvasComponents: {}
      });
      self.pages.set(pageId, page);
      return page;
    }
```

```
  }))
  .views(self => ({
    get breakpointsArray(): BreakpointType[] {
      return Array.from(self.breakpoints.values())
        .sort((a, b) => a.minWidth - b.minWidth);
    },

    get pagesArray(): PageModelType[] {
      return Array.from(self.pages.values());
    }
  }));

// src/models/PageModel.ts
const PageModel = types
  .model('Page', {
    id: types.identifier,
    slug: types.string,
    metadata: PageMetadataModel,
    rootComponent: types.late(() => ComponentModel),
    rootCanvasComponents: types.optional(types.map(ComponentModel), {}),
    createdAt: types.optional(types.Date, () => new Date()),
    updatedAt: types.optional(types.Date, () => new Date()),
  })
  .actions(self => ({
    // Root canvas component management (floating elements)
    addRootCanvasComponent(component: ComponentInstance |
       ComponentSnapshotIn) {
      self.rootCanvasComponents.set(component.id, component);
      self.updatedAt = new Date();
    },

    removeRootCanvasComponent(componentId: string) {
      self.rootCanvasComponents.delete(componentId);
      self.updatedAt = new Date();
    },

    updateRootCanvasComponent(componentId: string, updates: Partial<
       ComponentSnapshotIn>) {
      const component = self.rootCanvasComponents.get(componentId);
      if (component) {
        Object.assign(component, updates);
        self.updatedAt = new Date();
      }
    }
  }))
  .views(self => ({
    get rootCanvasComponentsArray(): ComponentInstance[] {
      return Array.from(self.rootCanvasComponents.values());
    },

    get visibleRootCanvasComponents(): ComponentInstance[] {
      return this.rootCanvasComponentsArray.filter(comp => comp.
         canvasVisible);
    },
```

```
103
104     get url(): string {
105       return `/${self.slug}`;
106     }
107   }));
108
109 export default PageModel;
110 export type PageModelType = Instance<typeof PageModel>;
```

Listing 6.4: Project and Page Models Implementation

6.2 High-Performance Canvas System

The canvas implementation represents the most performance-critical component of the entire system, requiring innovative approaches to maintain smooth 60fps interactions while handling potentially thousands of visual elements. The solution combines several advanced techniques: direct DOM manipulation to bypass React's virtual DOM for high-frequency operations, GPU-optimized CSS transforms to leverage hardware acceleration, and a sophisticated subscription system that decouples performance-critical updates from React's reconciliation process.

The implementation challenges traditional React patterns by recognizing that certain operations, particularly canvas transforms during pan and zoom, occur at frequencies that make React's reconciliation process a performance bottleneck. The solution maintains React compatibility for the overall application architecture while strategically bypassing React for the most performance-sensitive operations.

6.2.1 Transform Context Architecture

The transform context system represents a novel approach to state management that combines the benefits of React Context with the performance characteristics of direct DOM manipulation. The system stores transform state in a mutable ref rather than React state, eliminating unnecessary re-renders while providing a subscription mechanism for components that need to respond to transform changes.

```tsx
// src/contexts/TransformContext.tsx
import React, { createContext, useContext, useRef, MutableRefObject } from 'react';

interface CanvasTransform {
  zoom: number;   // Current zoom level (1.0 = 100%)
  panX: number;   // Horizontal pan offset in pixels
  panY: number;   // Vertical pan offset in pixels
}

interface TransformContextValue {
  // Transform state as ref (no React re-renders when changed)
  state: MutableRefObject<CanvasTransform>;
  // Subscribe to transform updates
  subscribe: (callback: () => void) => () => void;
}

const TransformContext = createContext<TransformContextValue | null>(null);

/**
 * TransformProvider - High-performance canvas transform state management
 *
 * Key Design Decisions:
 * 1. Transform state stored in useRef (mutable, no re-renders)
 * 2. Subscribers stored in useRef Set (stable across renders)
 * 3. Subscribe function returns unsubscribe callback (React useEffect pattern)
 * 4. Canvas calls notifySubscribers() after each transform update
 *
 * Performance Benefits:
 * - Zero React re-renders during transform operations
 * - Direct DOM updates for overlays (no virtual DOM diffing)
 * - Minimal memory allocations (reused refs and callbacks)
 * - Efficient subscription management (Set for O(1) add/remove)
 */
export const TransformProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
  // Transform state as ref - single source of truth for canvas transforms
  const transformState = useRef<CanvasTransform>({
    zoom: 1,    // Start at 100% zoom
    panX: 0,    // Start at origin
    panY: 0
  });

  // Set of subscriber callbacks - called when transform changes
  const subscribers = useRef<Set<() => void>>(new Set());

  /**
   * Subscribe to transform updates
   * @param callback - Function to call when transform changes
   * @returns Unsubscribe function for cleanup
   */
```

```
50    const subscribe = (callback: () => void): (() => void) => {
51      subscribers.current.add(callback);
52
53      // Return unsubscribe function for React useEffect cleanup
54      return () => {
55        subscribers.current.delete(callback);
56      };
57    };
58
59    /**
60     * Notify all subscribers of transform changes
61     * Called by Canvas after each transform update (pan, zoom, etc.)
62     */
63    const notifySubscribers = () => {
64      subscribers.current.forEach(callback => callback());
65    };
66
67    const contextValue: TransformContextValue = {
68      state: transformState,
69      subscribe
70    };
71
72    // Expose notifySubscribers for Canvas to call
73    (contextValue as any).notifySubscribers = notifySubscribers;
74
75    return (
76      <TransformContext.Provider value={contextValue}>
77        {children}
78      </TransformContext.Provider>
79    );
80  };
81
82  /**
83   * Hook to access transform context
84   */
85  export const useTransformContext = () => {
86    const context = useContext(TransformContext);
87    if (!context) {
88      throw new Error('useTransformContext must be used within a
         TransformProvider');
89    }
90    return context;
91  };
92
93  /**
94   * Hook to get the notifySubscribers function (for Canvas use)
95   */
96  export const useTransformNotifier = () => {
97    const context = useContext(TransformContext);
98    if (!context) {
99      throw new Error('useTransformNotifier must be used within a
         TransformProvider');
100   }
101   return (context as any).notifySubscribers;
```

```
};
```

Listing 6.5: High-Performance Transform Context Implementation

6.2.2 Canvas Component Implementation

The canvas component implementation demonstrates how performance-critical React components can be architected to achieve native-level performance while maintaining React's declarative programming model. The component uses direct DOM manipulation for transform operations while preserving React patterns for component lifecycle management and event handling.

The key innovation lies in separating high-frequency operations (transform updates) from low-frequency operations (component mounting, event handler attachment). This separation allows the system to achieve sub-millisecond update times for canvas operations while maintaining the developer experience benefits of React's component model.

```
// src/components/Canvas.tsx
import React, { useRef, useCallback, useEffect } from 'react';
import { useTransformContext, useTransformNotifier } from '@/contexts/
    TransformContext';
import { observer } from 'mobx-react-lite';
import { useStore } from '@/hooks/useStore';

const CanvasInner = observer(() => {
  // Performance-optimized refs for direct DOM manipulation
  const groundRef = useRef<HTMLDivElement>(null);  // Viewport container
  const cameraRef = useRef<HTMLDivElement>(null);  // Transform target
  const isDragging = useRef(false);
  const lastMousePos = useRef({ x: 0, y: 0 });
  const rootStore = useStore();

  // Get transform state and notifier from context
  const { state: transformState } = useTransformContext();
  const notifySubscribers = useTransformNotifier();

  /**
   * Apply transform to canvas camera element and notify subscribers
   *
   * Performance optimizations:
   * - Direct DOM manipulation (no React re-render)
   * - String interpolation (faster than template literals in loops)
   * - Single transform application (not separate translate/scale)
   * - Cached callback with stable dependencies
   */
  const applyTransform = useCallback(() => {
    if (cameraRef.current) {
      const { zoom, panX, panY } = transformState.current;

      // Direct CSS transform application - fastest approach
      const transformString = `translate(${panX}px, ${panY}px) scale(${
```

```
        zoom})`;
      cameraRef.current.style.transform = transformString;

      // Update data attribute for debugging and external access
      cameraRef.current.setAttribute('data-camera-transform', `${panX},${
        panY},${zoom}`);

      // Notify all subscribers (HudSurface, debug overlays, etc.)
      notifySubscribers();
    }
  }, [notifySubscribers, transformState]);

  // Mouse wheel handling - pan by default, zoom with Command/Ctrl key
  const handleWheel = useCallback((e: WheelEvent) => {
    e.preventDefault();

    const isZoomModifier = e.metaKey || e.ctrlKey;

    if (isZoomModifier) {
      // Cursor-centered zoom algorithm
      const rect = groundRef.current!.getBoundingClientRect();
      const mouseX = e.clientX - rect.left;
      const mouseY = e.clientY - rect.top;

      const scaleFactor = 1.1;
      const zoomDirection = e.deltaY > 0 ? -1 : 1;
      const factor = Math.pow(scaleFactor, zoomDirection);
      const newZoom = Math.max(0.1, Math.min(5, transformState.current.
        zoom * factor));

      // Convert mouse position to world coordinates
      const worldX = (mouseX - transformState.current.panX) /
        transformState.current.zoom;
      const worldY = (mouseY - transformState.current.panY) /
        transformState.current.zoom;

      // Apply zoom while keeping mouse position stable
      const newPanX = mouseX - worldX * newZoom;
      const newPanY = mouseY - worldY * newZoom;

      transformState.current.zoom = newZoom;
      transformState.current.panX = newPanX;
      transformState.current.panY = newPanY;
    } else {
      // Pan mode - direct manipulation
      transformState.current.panX -= e.deltaX;
      transformState.current.panY -= e.deltaY;
    }

    applyTransform();
  }, [applyTransform, transformState]);

  // Mouse drag handling for pan operations
  const handleMouseDown = useCallback((e: MouseEvent) => {
```

```
    if (e.button !== 1 && !(e.button === 0 && e.spaceKey)) return; // Middle mouse or space+click

    e.preventDefault();
    isDragging.current = true;
    lastMousePos.current = { x: e.clientX, y: e.clientY };

    document.addEventListener('mousemove', handleMouseMove);
    document.addEventListener('mouseup', handleMouseUp);
  }, []);

  const handleMouseMove = useCallback((e: MouseEvent) => {
    if (!isDragging.current) return;

    const deltaX = e.clientX - lastMousePos.current.x;
    const deltaY = e.clientY - lastMousePos.current.y;

    transformState.current.panX += deltaX;
    transformState.current.panY += deltaY;

    lastMousePos.current = { x: e.clientX, y: e.clientY };
    applyTransform();
  }, [applyTransform, transformState]);

  const handleMouseUp = useCallback(() => {
    isDragging.current = false;
    document.removeEventListener('mousemove', handleMouseMove);
    document.removeEventListener('mouseup', handleMouseUp);
  }, [handleMouseMove]);

  // Event listener setup
  useEffect(() => {
    const ground = groundRef.current;
    if (!ground) return;

    ground.addEventListener('wheel', handleWheel, { passive: false });
    ground.addEventListener('mousedown', handleMouseDown);

    return () => {
      ground.removeEventListener('wheel', handleWheel);
      ground.removeEventListener('mousedown', handleMouseDown);
    };
  }, [handleWheel, handleMouseDown]);

  return (
    <div className="relative flex-1 overflow-hidden bg-gray-100">
      {/* Canvas ground - viewport container */}
      <div
        ref={groundRef}
        className="absolute inset-0 cursor-grab active:cursor-grabbing"
        style={{ touchAction: 'none' }}
      >
        {/* Canvas camera - transform target */}
        <div
```

```
136             ref={cameraRef}
137             className="absolute top-0 left-0 w-0 h-0 origin-top-left will-
                    change-transform"
138             style={{
139               transform: 'translate(0px, 0px) scale(1)',
140               isolation: 'isolate',
141               contain: 'layout style'
142             }}
143           >
144             {/* Rendered content goes here */}
145             <ResponsivePageRenderer />
146           </div>
147         </div>
148
149         {/* Canvas toolbar */}
150         <div className="absolute top-4 left-4">
151           <Toolbar />
152         </div>
153       </div>
154     );
155 });
156
157 // Main Canvas component wrapped with TransformProvider context
158 const Canvas = () => <CanvasInner />;
159
160 export default Canvas;
```

Listing 6.6: Optimized Canvas Implementation

6.3 Component Rendering and Meta-Programming

The component rendering system represents the culmination of the meta-programming approach, transforming serializable component data structures into fully functional React elements. This system bridges the gap between the visual editor's data-driven component definitions and the runtime requirements of React applications.

The rendering implementation addresses several complex challenges: dynamic component type resolution that can handle both built-in HTML elements and custom React components, responsive property resolution that adapts component behavior based on breakpoint contexts, and event handler injection that enables editor functionality without compromising component behavior.

6.3.1 Dynamic Component Generation

The core rendering engine employs React.createElement to dynamically generate components from MST data structures. This approach enables real-time component modification while preserving full React functionality, including proper event handling, lifecycle management, and state synchro-

nization.

```tsx
// src/components/ComponentRenderer.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import { ComponentInstance, BreakpointType } from '@/models';
import { useStore } from '@/hooks/useStore';
import { EditorTool } from '@/stores/EditorUIStore';

interface ComponentRendererProps {
  component: ComponentInstance;
  breakpointId: string;
  allBreakpoints: BreakpointType[];
  primaryId: string;
}

const ComponentRenderer = observer(({
  component,
  breakpointId,
  allBreakpoints,
  primaryId
}: ComponentRendererProps) => {
  const { editorUI } = useStore();

  // Resolve responsive properties for current breakpoint
  const { attributes, style } = component.getResolvedProps(
    breakpointId,
    allBreakpoints,
    primaryId
  );

  // Build final props for React element
  const finalProps: Record<string, unknown> = {
    ...attributes,
    style: Object.keys(style).length ? style : undefined,
    'data-component-id': `${breakpointId}-${component.id}`,

    // Add click handler for selection in editor mode
    onClick: (e: React.MouseEvent) => {
      e.stopPropagation();
      if (editorUI.selectedTool === EditorTool.SELECT) {
        editorUI.selectComponent(component, breakpointId);
      }
    }
  };

  // Recursively render children
  const children = component.children.map(child =>
    <ComponentRenderer
      key={child.id}
      component={child}
      breakpointId={breakpointId}
      allBreakpoints={allBreakpoints}
      primaryId={primaryId}
```

```
    />
  );

  // Handle host elements (div, button, img, etc.)
  if (component.isHostElement) {
    return React.createElement(
      component.type,
      finalProps,
      children.length ? children : attributes.children
    );
  }

  // Handle custom function components
  const CustomComponent = (window as any).__componentRegistry?.[component.
     type];
  if (CustomComponent) {
    return <CustomComponent {...finalProps}>{children}</CustomComponent>;
  }

  // Fallback for unknown components
  return (
    <div
      style={{
        border: '1px dashed orange',
        padding: 8,
        backgroundColor: 'rgba(255, 165, 0, 0.1)'
      }}
    >
      <div style={{ fontSize: 12, color: 'orange', marginBottom: 4 }}>
        Unknown component: {component.type}
      </div>
      {children}
    </div>
  );
});

export default ComponentRenderer;
```

Listing 6.7: Recursive Component Renderer Implementation

6.3.2 GroundWrapper Implementation

The GroundWrapper component represents a critical architectural innovation that enables Framer-style canvas positioning while maintaining React component patterns. Rather than applying transforms to individual elements, the system wraps each canvas element in a positioning container that handles coordinate transformation, scaling, and rotation independently.

This approach provides several advantages over traditional canvas implementations: each element maintains its own transform state, enabling independent manipulation without affecting other elements; the positioning system scales efficiently with the number of elements; and the wrapper

abstraction simplifies the integration of complex positioning logic with React's component model.

```tsx
// src/components/GroundWrapper.tsx
import React, { forwardRef } from 'react';
import { observer } from 'mobx-react-lite';

interface GroundWrapperProps {
  id: string;
  x: number;
  y: number;
  scale?: number;
  rotation?: number;
  zIndex?: number;
  width?: number;
  height?: number;
  className?: string;
  children: React.ReactNode;
  onClick?: (e: React.MouseEvent) => void;
  visible?: boolean;
}

/**
 * GroundWrapper - Individual positioning wrapper for canvas elements
 *
 * Replicates Framer's groundNodeWrapper approach where each element
 * on the canvas has its own positioning wrapper with independent
 * transform state, rather than using a single camera transform.
 */
const GroundWrapper = observer(forwardRef<HTMLDivElement,
    GroundWrapperProps>(
  function GroundWrapper({
    id, x, y, scale = 1, rotation = 0, zIndex = 0,
    width, height, className = '', children, onClick, visible = true
  }, ref) {

    // Build CSS transform string
    const transform = `translate(${x}px, ${y}px) scale(${scale}) rotate(${
        rotation}deg)`;

    // Inline styles for maximum performance
    const style: React.CSSProperties = {
      position: 'absolute',
      top: 0, left: 0,
      transform,
      transformOrigin: 'left top',  // Critical for predictable transforms
      willChange: 'transform',       // GPU optimization
      isolation: 'isolate',          // Create stacking context
      contain: 'layout style',       // Performance optimization
      zIndex,
      display: visible ? 'block' : 'none',
      width: width ? `${width}px` : 'auto',
      height: height ? `${height}px` : 'auto',
    };

```

```
    return (
      <div
        ref={ref}
        id={`ground-wrapper-${id}`}
        className={`ground-wrapper ${className}`}
        style={style}
        onClick={onClick}
        data-ground-wrapper-id={id}
      >
        {/* Inner container for content */}
        <div
          id={`ground-inner-${id}`}
          className="ground-inner"
          style={{
            transformOrigin: 'left top',
            width: '100%', height: '100%',
            overflow: 'visible'
          }}
        >
          {children}
        </div>
      </div>
    );
  }
));

export default GroundWrapper;
```

Listing 6.8: GroundWrapper Implementation

6.4 Selection Overlay System

The selection overlay system represents one of the most technically challenging aspects of the implementation, requiring precise coordinate transformation between multiple coordinate systems while maintaining 60fps performance during canvas navigation. The system must handle both traditional DOM elements (which exist within the document flow) and virtual canvas elements (which exist in canvas coordinate space).

The implementation adopts Framer's HudSurface approach, where selection overlays exist as fixed-position elements that are updated through direct DOM manipulation rather than React re-renders. This approach ensures that overlay updates remain synchronized with canvas transforms without introducing performance bottlenecks.

6.4.1 Coordinate System Management

The HudSurface system manages the complex challenge of maintaining accurate overlay positioning across two fundamentally different coordinate systems. DOM elements within breakpoint viewports can be positioned using standard getBoundingClientRect() measurements, while floating canvas elements require mathematical transformation from canvas coordinates to screen coordinates.

The coordinate transformation algorithm accounts for canvas zoom, pan offset, and the canvas container's position within the browser viewport, ensuring pixel-perfect overlay alignment regardless of canvas navigation state.

```
// src/components/HudSurface.tsx
import React, { useRef, useEffect, useCallback, useState } from 'react';
import { observer } from 'mobx-react-lite';
import { useStore } from '@/hooks/useStore';
import { useTransformContext } from '@/contexts/TransformContext';
import { EditorTool } from '@/stores/EditorUIStore';

const HudSurface = observer(() => {
  const { editorUI } = useStore();
  const { state: transformState, subscribe } = useTransformContext();

  // Canvas container rect cached to prevent repeated DOM queries
  const [canvasContainerRect, setCanvasContainerRect] = useState<DOMRect |
      null>(null);
  const overlayRef = useRef<HTMLDivElement>(null);

  // Cache canvas container position
  useEffect(() => {
    const updateCanvasRect = () => {
      const canvasContainer = document.querySelector('[data-canvas-
          container]') as HTMLElement;
      if (canvasContainer) {
        setCanvasContainerRect(canvasContainer.getBoundingClientRect());
      }
    };

    updateCanvasRect();
    window.addEventListener('resize', updateCanvasRect);
    return () => window.removeEventListener('resize', updateCanvasRect);
  }, []);

  /**
   * Update overlay position via direct DOM manipulation
   * Handles both real DOM elements and virtual canvas elements
   */
  const updateOverlayPosition = useCallback(() => {
    if (!overlayRef.current || !canvasContainerRect) return;

    const overlay = overlayRef.current;
    const { panX, panY, zoom } = transformState.current;
```

```
39
40      // Hide overlay if not in select mode
41      if (editorUI.selectedTool !== EditorTool.SELECT) {
42        overlay.style.display = 'none';
43        return;
44      }
45
46      // Case 1: Real DOM elements (components within breakpoints)
47      if (editorUI.selectedComponent && editorUI.selectedBreakpoint) {
48        const breakpointComponentId = `${editorUI.selectedBreakpoint.id}-${
              editorUI.selectedComponent.id}`;
49        const element = document.querySelector(
50          `[data-component-id="${breakpointComponentId}"]`
51        ) as HTMLElement;
52
53        if (element) {
54          const rect = element.getBoundingClientRect();
55
56          // Direct screen coordinates (element already positioned correctly
                )
57          const x = rect.left - 2;  // -2px for border offset
58          const y = rect.top - 2;
59
60          overlay.style.display = 'block';
61          overlay.style.transform = `translate(${x}px, ${y}px)`;
62          overlay.style.width = `${rect.width + 4}px`;
63          overlay.style.height = `${rect.height + 4}px`;
64          return;
65        }
66      }
67
68      // Case 2: Virtual canvas elements (floating components)
69      if (editorUI.selectedRootCanvasComponent?.isRootCanvasComponent) {
70        const bounds = editorUI.selectedRootCanvasComponent.canvasBounds;
71        if (bounds) {
72          // Transform canvas coordinates to screen coordinates
73          // Formula: screenPos = (canvasPos * zoom) + pan + containerOffset
74          const screenX = (bounds.x * zoom) + panX + canvasContainerRect.
                left;
75          const screenY = (bounds.y * zoom) + panY + canvasContainerRect.top
                ;
76
77          const x = screenX - 2;
78          const y = screenY - 2;
79          const scaledWidth = bounds.width * zoom;
80          const scaledHeight = bounds.height * zoom;
81
82          overlay.style.display = 'block';
83          overlay.style.transform = `translate(${x}px, ${y}px)`;
84          overlay.style.width = `${scaledWidth + 4}px`;
85          overlay.style.height = `${scaledHeight + 4}px`;
86          return;
87        }
88      }
```

```
89
90      // No valid selection - hide overlay
91      overlay.style.display = 'none';
92    }, [editorUI, canvasContainerRect, transformState]);
93
94    // Subscribe to transform updates (high-frequency, no React re-renders)
95    useEffect(() => {
96      const unsubscribe = subscribe(() => {
97        updateOverlayPosition(); // Direct DOM update
98      });
99      return unsubscribe;
100   }, [subscribe, updateOverlayPosition]);
101
102   // React updates (low-frequency, only when selection changes)
103   useEffect(() => {
104     updateOverlayPosition();
105   }, [updateOverlayPosition]);
106
107   return (
108     <div className="fixed inset-0 pointer-events-none" style={{ zIndex: 10
          }}>
109       {/* Single unified selection overlay */}
110       <div
111         ref={overlayRef}
112         className="absolute pointer-events-none border-2 border-blue-500
             bg-blue-500/10"
113         style={{ display: 'none' }}
114       >
115         {/* Selection handles */}
116         <div className="absolute -top-1 -left-1 w-2 h-2 bg-blue-500
             rounded-full" />
117         <div className="absolute -top-1 left-1/2 transform -translate-x
             -1/2 w-2 h-2 bg-blue-500 rounded-full" />
118         <div className="absolute -top-1 -right-1 w-2 h-2 bg-blue-500
             rounded-full" />
119         <div className="absolute top-1/2 -left-1 transform -translate-y
             -1/2 w-2 h-2 bg-blue-500 rounded-full" />
120         <div className="absolute top-1/2 -right-1 transform -translate-y
             -1/2 w-2 h-2 bg-blue-500 rounded-full" />
121         <div className="absolute -bottom-1 -left-1 w-2 h-2 bg-blue-500
             rounded-full" />
122         <div className="absolute -bottom-1 left-1/2 transform -translate-x
             -1/2 w-2 h-2 bg-blue-500 rounded-full" />
123         <div className="absolute -bottom-1 -right-1 w-2 h-2 bg-blue-500
             rounded-full" />
124       </div>
125     </div>
126   );
127 });
128
129 export default HudSurface;
```

Listing 6.9: HudSurface Implementation

6.5 Performance Validation and Analysis

The performance validation process demonstrates that the implementation successfully achieves the ambitious performance targets established in the requirements analysis. The systematic optimization of critical paths, combined with innovative architectural patterns, enables the system to maintain professional-grade responsiveness even under intensive usage scenarios.

The performance measurement methodology focused on the most demanding operations: canvas navigation with hundreds of elements, rapid component selection changes, real-time property updates, and complex component tree manipulations. Each operation was measured across multiple scenarios to ensure consistent performance characteristics.

6.5.1 Performance Metrics

The performance analysis reveals significant improvements over traditional React-based visual editors:

Operation	Target	Achieved	Optimization Technique
Canvas Pan/Zoom	<16.67ms (60fps)	0.1-0.5ms	Direct DOM + GPU acceleration
Component Selection	<100ms	10-30ms	Cached DOM queries + MobX
Property Updates	Real-time	<10ms	Fine-grained reactivity
Tree Reordering	Smooth	<100ms	MST snapshots + React keys
Overlay Updates	60fps sync	0.2-0.8ms	Subscription pattern

Table 6.1: Performance measurements compared to initial targets

The results demonstrate that the implementation not only meets but significantly exceeds the established performance targets. Canvas operations achieve sub-millisecond update times, representing a 30-50x improvement over traditional React-based approaches. Component selection and property updates maintain real-time responsiveness, while complex operations such as tree reordering remain well within acceptable performance boundaries.

6.5.2 Architectural Success Validation

The implementation successfully validates all four core architectural principles established in the design phase. The unified component model demonstrates its effectiveness by seamlessly handling both hierarchical and floating elements through a single data structure and rendering pipeline. The performance-first canvas architecture achieves O(1) complexity for pan and zoom operations, maintaining constant performance regardless of element count.

The meta-programming approach proves its viability by successfully transforming component configurations into functional React elements while preserving full component behavior and interactivity. The reactive state management system demonstrates the ability to handle both editor-specific updates and runtime component behavior without performance degradation or architectural conflicts.

These validation results confirm that the research prototype successfully addresses the core challenges identified in the problem analysis, providing a solid foundation for future development and potential commercial application.

7. Presentation of Results

[Description of the results from all preceding chapters as well as the previously generated result artifacts with evaluation of how these are to be classified]

8. Summary

[Aggregated retrospective brief description of the work]

8.1 Conclusions

[Description of the overall conclusions to be noted in connection with the work]

8.2 Limitations

[Description of the results of a critical reflection and justification of what the work cannot achieve]

8.3 Outlook

[Description and justification of potential future follow-up activities in connection with your work (e.g., further requirements, theory building, ...)]

References

A. Appendix

A.1 Source Code

[Extended source code examples]

A.2 Additional Figures

Include supplementary figures or tables.

A.3 Survey Instruments

Include questionnaires or interview guides if applicable.

A.4 Tips for Writing Your Thesis

- Use neutral, professional language. Avoid first person ("I").
- Cite credible and worthy sources (e.g., scientific articles and textbooks; avoid blogs and never cite Wikipedia[1]).
- Cite correctly and consistently.
- Do not use footnotes for literature references.
- Research thoroughly the state of science and technology.
- Pay attention to the quality of the work (e.g., spelling).
- Inform yourself in advance about how to work and write scientifically.
- Use LaTeX[2].

[1] Wikipedia itself recommends avoiding citation of Wikipedia content in academic contexts.
[2] No support provided for installation, use and adaptation of LaTeXtemplates!

Academic integrity declaration

I hereby declare that I have written all parts of this thesis independently and that I have not used any sources other than those indicated in the thesis and that I have not submitted the thesis in the same or a similar form for any other examination. All verbatim or analogous copies and quotations have been identified and verified.

Information on the use of AI-based tools

I affirm that I have not used any AI-based tools whose use has been explicitly excluded in writing by the examiner. I am aware that the use of texts or other content and products generated by AI-based tools does not guarantee their quality. I am fully responsible for the adoption of any machine-generated passages used by me and bear the responsibility for any erroneous or distorted content, faulty references, violations of data protection and copyright law or plagiarism generated by the AI. I also affirm that my creative influence predominates in the present work.

Berlin, August 26, 2025

_____

Your Full Name