Progressive SSR Framework: Ziro

Nariman Movaffaghi
Supervisor: Soroush Vedaei

December 1, 2024

Contents

1 Abstract **4**

2 Introduction **5**

3 Literature Review **7**

3.1 Evolution of Web Technologies 7

3.2 The Rise of Server-Side Rendering (SSR) Frameworks 7

3.3 Review of Key SSR Frameworks 8

3.4 The Market Landscape of SSR Frameworks 10

3.5 Why Market Metrics Matter 11

3.6 Evolution of Hosting Solutions and Their Impact on SSR 11

3.7 Ziro's Position: A Foundation for Modern Web Development . . 12

3.8 Conclusion 14

4 Methodology **16**

4.1 Building the Core of Ziro 16

4.2 Creating `ziro/router` 16

4.3 Using Test-Driven Development 16

4.4 Moving to `ziro/react` 17

4.5 Workflow for Development 17

4.6 Improving Type Safety with TypeScript 17

4.7 Adding Server-Side Compatibility 18

4.8 Improving and Refining the Framework 18

4.9 Results of the Development Process 18

5 Theses **19**

6 Technical Specifications **20**

7 Results **21**

8 Future Works **22**

1 Abstract

Web development has evolved to require frameworks that are both flexible and efficient. Existing server-side rendering (SSR) frameworks, such as Next.js and Nuxt.js, offer powerful solutions but can be less suited for building modular systems with pre-integrated features like authentication and administrative tools. This project presents Ziro, an SSR framework designed to explore the feasibility of creating a plugin-driven architecture within the JavaScript ecosystem.

Ziro is built to provide a practical foundation for developing dynamic web applications. Its modular structure allows for the addition of pre-built plugins, aiming to reduce repetitive work for developers. Key components like the `ziro/router` ensure compatibility across environments, while the `ziro/generator` supports type safety through automated TypeScript type generation. Ziro also integrates server-side and client-side rendering to improve both performance and user experience.

The framework has been developed iteratively, with a focus on testing edge cases using Vitest and refining features based on practical needs. By combining these elements, Ziro seeks to address common development challenges while providing a basis for future improvements. This proposal outlines its development process and the lessons learned from building a modular web framework.

2 Introduction

Web development frameworks have evolved significantly in recent years, yet many still lack a cohesive approach to combining scalability, modularity, and developer-friendly abstractions. Frameworks like Next.js and Remix provide powerful tools for building web applications, but they do not expose certain critical layers, such as the router, for plugin-based customization. This limitation often forces developers to repeatedly create essential features like authentication, which involves setting up routes, middleware, actions, and components manually for every project.

Ziro addresses this gap by introducing a thoughtful approach that abstracts multiple parts of a full-stack framework while allowing them to scale independently. Ziro enables plugins to dynamically add custom pages, endpoints, middleware, and actions. This approach significantly reduces repetitive work and accelerates the development process, making it ideal for tasks like implementing reusable authentication systems or integrating predefined pages across projects.

Additionally, Ziro adheres to web standards to ensure compatibility across various runtimes, including Node.js and browsers. The framework is structured into separate packages to maintain modularity and framework-agnostic design:

- ziro/router: Contains the core routing concepts, designed to work independently of any specific framework.
- ziro/react: A React-compatible implementation for seamless integration with React-based projects.
- ziro/generator: A tool for generating TypeScript type-safe definitions to ensure robust developer workflows.

Ziro's features are grounded in modern web development principles. It offers a fully typesafe routing system, ensuring that all actions and data derived from middleware or layouts maintain type safety throughout the application. With its sequential data flow, loaders from nested pages run in order, allowing data fetched at higher levels to cascade to lower ones. The plugin system further enhances flexibility by simplifying the process of extending applications with custom functionalities.

Targeted primarily at developers, Ziro also opens the door to building tools for broader audiences, such as WordPress users who may not have a strong technical background. By introducing modularity and preconfigured components, it streamlines the creation of dynamic, scalable applications like e-commerce platforms, personal blogs, and full-stack applications.

Developing Ziro has presented several challenges, including implementing robust type safety—a feature still relatively novel in modern frameworks—and ensuring seamless integration across different rendering modes: SSR, CSR, and a partially SSR mode that streams data to users. The result is a proof of concept that not only demonstrates the feasibility of combining full-stack JavaScript apps with plugin-based scaling but also provides a modular foundation for scalable, dynamic application development.

This introduction outlines Ziro's core philosophy, features, and potential impact, setting the stage for a deeper exploration of its architecture, methodology, and applications.

3 Literature Review

3.1 Evolution of Web Technologies

In the early days of web development, frameworks and technologies like PHP and Ruby on Rails were popular, allowing developers to deliver full HTML pages generated on the server. These server-rendered applications had clear benefits—users received complete, pre-generated content, making initial page loads quick and ensuring that content was easy for search engines to index. This method effectively addressed early concerns around user experience and web visibility.

However, things changed with the introduction of Single Page Applications (SPAs). SPAs revolutionized development by providing smoother client-side experiences. They shifted the server's role to mainly delivering static assets while JavaScript managed the content rendering on the client side. This model was attractive due to its improved maintainability and streamlined development. Frameworks like Angular[1]. and later React[2]. and Vuejs[3] became popular for their client-side rendering (CSR) that delivered a richer and more dynamic user experience.

While SPAs brought many advantages, they also had their challenges. One major weakness of SPAs was their initial page load performance. Since the server only served a minimal HTML shell and relied on JavaScript to render content, the time taken for the browser to download and execute the JavaScript bundle impacted the initial load speed. This delay affected user experience and search engine optimization (SEO), as search engines struggled to index content that was rendered dynamically in the browser. To address these issues, SSR frameworks emerged as a practical solution.

3.2 The Rise of Server-Side Rendering (SSR) Frameworks

SSR frameworks have changed the landscape of web development, offering benefits that meet the increasing demand for better user experiences and search engine optimization (SEO). These frameworks pre-render content on the server, allowing for faster initial page loads, improved discoverability by search engines, and a smoother user experience. With SSR, the server processes the requests and delivers fully rendered HTML pages to the client, reducing the browser's burden of generating content. This approach has proven valuable as applications have become more interactive and data-heavy.

[1] https://angularjs.org/
[2] https://react.dev/
[3] https://vuejs.org

A significant advantage of SSR frameworks is their positive impact on Core Web Vitals[4], a set of essential metrics used to measure user experience. These include:

- **Largest Contentful Paint (LCP)**: Measures loading performance; SSR improves LCP by ensuring that the initial content loads quickly.
- **First Input Delay (FID)**: Evaluates interactivity; SSR can reduce FID by delivering fully rendered pages that are interactive sooner.
- **Cumulative Layout Shift (CLS)**: Assesses visual stability; SSR reduces layout shifts by ensuring content is structured and rendered on the server before the client receives it.

By improving initial page load times, SSR frameworks contribute to better performance in Core Web Vitals, which directly impacts user satisfaction and search engine ranking.

Another benefit of SSR is the ability to generate dynamic Open Graph[5] (OG) tags before the HTML is sent to the user. OG tags are crucial for sharing rich and structured content on social media platforms, enhancing how a web page appears when shared. These tags must exist in the initial response for social media crawlers to read them and display previews accurately. This ensures that shared content looks appealing and informative.

Overall, the evolution from traditional server-rendered pages to SPAs and back to SSR frameworks has shaped modern web development. While SPAs have their benefits, particularly in terms of client-side interactivity and reduced server load, SSR frameworks offer practical solutions for better SEO, improved performance, and enhanced user experience. By implementing SSR, developers can leverage server-side processing to deliver fully rendered pages with optimal loading times, making web applications faster and more accessible.

3.3 Review of Key SSR Frameworks

Next.js[6] has become one of the most widely adopted SSR frameworks within the JavaScript ecosystem. Built around React, it offers a strong set of features, including file-based routing, built-in SSR capabilities, and API routes. It provides developers flexibility through hybrid rendering options that combine both SSR and Static Site Generation (SSG). Next.js supports two different routing

[4] https://web.dev/articles/vitals#core-web-vitals
[5] https://ogp.me/
[6] https://nextjs.org/

systems: the Page Router[7] and the App Router[8]. The Page Router has limited TypeScript support, which can lead to challenges in maintaining type safety. In contrast, the App Router offers much better TypeScript integration, making it easier for developers to write type-safe code. However, while Next.js excels in out-of-the-box features, it lacks a comprehensive plugin system for extensive customization. This can make large-scale projects more complex as developers need to extend all middleware rules in a single file and create routes manually, resulting in code repetition and in some cases inefficiency.

Remix[9] which was made on top of **React Router**[10] by developers behind it, provides a unique approach to handling routing and data fetching. It focuses heavily on nested routing and uses loaders for efficient data management. This model improves data handling within complex page structures. Remix by introducing delightful APIs for the data fetching flow has improved its developer experience. However, lack of middleware having multiple actions on the same route can still be manual, leading to more boilerplate code. Additionally, Remix doesn't have an easy solution for type-safety, but it works with better developer experience than Next.js in terms of routing and data fetching.

Nuxt.js[11] is the go-to SSR framework for Vue.js applications and offers similar capabilities to Next.js, but tailored for the Vue ecosystem. It has built-in SSR, a modular structure, and supports customizability. In compared to Next.js and Remix, Nuxt offers more APIs to customize the logic behind the app. Of course modifying or extending core logic requires a deep understanding of the framework's internals but Nuxt has a more comprehensive plugin system than Next.js and Remix which makes it easier to extend the core functionalities. Additionally, Nuxt.js has great support for TypeScript, making it easier for developers to write type-safe code and improve maintainability.

Astro[12] is a newer framework that has gained attention for its support of SSR and hybrid rendering, which blends SSR and Client-Side Rendering (CSR). Its minimal JavaScript footprint helps optimize performance by allowing developers to choose which frameworks to use for different parts of their project. Astro also has first-class support for TypeScript, making it easier for developers to write type-safe code. Although Astro's modular design provides significant customization, integrating various components can sometimes feel fragmented and may require developers to piece together solutions for a cohesive application. Additionally, Astro offers a revolutionary solution for handling server actions[13] and data validation, streamlining the process and enhancing developer productivity. One of the standout features of Astro is its framework-agnostic nature.

[7] https://nextjs.org/docs/pages
[8] https://nextjs.org/docs/app
[9] https://remix.run/
[10] https://reactrouter.com/
[11] https://nuxt.com
[12] https://astro.build/
[13] https://docs.astro.build/en/guides/actions/

Developers can write components using React, Vue.js, Svelte, Markdown, etc, allowing for greater flexibility and reusability of existing code. This capability makes Astro an attractive option for teams that work with multiple front-end technologies or are transitioning between frameworks. By supporting a variety of component models, Astro enables developers to leverage the strengths of different frameworks within a single project, optimizing performance and maintainability.

3.4 The Market Landscape of SSR Frameworks

The growth of SSR frameworks has been notable in recent years, driven by their potential to enhance user experience and SEO. In the context of market adoption, it's important to understand how frameworks have been received by the development community and the proportion they represent in web development projects.

Next.js stands out as the most widely used SSR framework, supported by its robust integration with React. According to the State of Frontend 2023[14] report, Next.js commands a significant share of the market, with over 56% of developers who work in web development using it for their SSR needs. The growth can be attributed to its ease of use, stability, and large community, which makes it a good choice for a wide range of projects.

Nuxt.js, continues to be a key player within the Vue.js ecosystem. While its adoption is smaller compared to Next.js, it remains a preferred choice for Vue developers. Its modular architecture and focus on simplifying SSR make it an efficient tool for building dynamic, server-rendered applications. According to the State of Frontend 2023 survey, almost 23% of developers prefer using Nuxt.js, and this percentage grows each year based on the charts. Nuxt.js has a strong presence in the market, with a significant number of developers leveraging its capabilities for SSR projects.

Remix, despite being newer than Next.js and Nuxt.js, has shown rapid growth in popularity, especially among developers seeking greater control over routing and data fetching. The State of Frontend 2023 survey mentioned that while Remix's adoption is still below that of Next.js and Nuxt.js, it has gained significant attention for its modern approach to data management.

Astro has emerged as a promising framework for projects prioritizing performance and reduced JavaScript load. According to the State of Frontend 2023 survey, Astro is used by more developers than Remix, highlighting its growing popularity. Its unique hybrid approach appeals to developers who want the flexibility of combining SSR and CSR efficiently. The use of Astro is increasing,

[14] https://2023.stateofjs.com/en-US/libraries/meta-frameworks/

particularly in projects where minimal JavaScript and high performance are key considerations.

3.5 Why Market Metrics Matter

Market size or community size alone is not always the most crucial factor when evaluating frameworks. Instead, their progress and the timing of their introduction reveal important insights. For instance, Next.js, launched in 2016, holds a substantial market share due to its established presence. Meanwhile, newer frameworks like Astro, introduced in 2021, have already captured nearly 19% of the market, reflecting their rapid adoption and appeal to developers. This suggests that Astro delivers features that resonate strongly with modern needs.

Rather than selecting a framework outright, we analyze usage metrics and growth trends to identify why these frameworks succeed. This evaluation helps us understand features that can be beneficial to Ziro. For example, Next.js demonstrates reliability with its long-term adoption, but newer frameworks like Astro showcase innovation and efficiency that might inspire Ziro's design.

3.6 Evolution of Hosting Solutions and Their Impact on SSR

The growth of server-side rendering (SSR) frameworks has been accompanied by significant advancements in hosting technologies. In the past, web applications were often hosted on virtual private servers (VPS), which, while flexible, required significant manual configuration and maintenance. This setup posed challenges in terms of scalability and performance, making it less suitable for SSR, which demands efficient server processing.

Recently, a shift has occurred with the emergence of modern serverless and edge hosting solutions. Platforms such as Vercel[15], Netlify[16], Cloudflare Pages[17], Deno Deploy[18], and Fly.io[19] have transformed how SSR frameworks are deployed. These services offer affordable, scalable hosting that makes it easier for developers to manage applications without worrying about infrastructure complexity.

The concept of servers-on-the-edge—where code runs closer to the end user—has been particularly impactful. Platforms like Cloudflare Pages , Deno Deploy, and

[15] https://vercel.com
[16] https://netlify.com
[17] https://pages.cloudflare.com/
[18] https://deno.com/deploy
[19] https://fly.io/deploy

Fly.io enable server-side code to be executed at various locations around the world. This reduces latency, enhances load times, and ultimately nullifies many of the drawbacks historically associated with SSR, such as slower response times and scalability issues. This accessibility and cost-effectiveness have further contributed to the adoption of SSR frameworks, making them a practical choice for developers aiming to create fast and efficient web applications.

3.7 Ziro's Position: A Foundation for Modern Web Development

The modern landscape of web development demands frameworks that not only streamline the development process but also empower developers with flexible and powerful tools. While existing frameworks like Next.js, Nuxt.js, and Astro offer impressive capabilities, they often lack the foundational elements necessary for building comprehensive, feature-rich web applications akin to the advanced content management systems (CMS) seen in the PHP and Laravel ecosystems, such as WordPress[20] and Laravel Nova[21]. This is where Ziro steps in.

The Vision Behind Ziro

The primary motivation for developing Ziro was to create a framework that enables developers to build applications with the simplicity and extensibility akin to established CMS platforms but within the JavaScript ecosystem. For instance, Laravel Nova is a powerful admin panel for Laravel that integrates seamlessly into the framework, providing a ready-to-use, configurable interface for managing data and content. By installing a plugin or package in Laravel, developers can extend their applications with new functionalities, such as authentication pages or admin dashboards, that are automatically configured through a single configuration file.

Ziro aims to bring this level of seamless integration and extensibility to the JavaScript world, enabling developers to build complex, dynamic web applications without having to reinvent the wheel for common features. With Ziro, adding essential components like authentication pages, administrative panels, or custom workflows should be as simple as including a package and configuring it through an intuitive, centralized settings file.

[20] https://wordpress.org
[21] https://nova.laravel.com/

Ziro's Key Features and Advantages

1. **Modular Plugin Architecture**: Ziro's most distinguishing feature is its modular plugin architecture. Unlike existing frameworks that often require developers to build complex middleware or extend core logic from scratch, Ziro allows developers to add new functionalities with pre-built plugins that can be installed and activated with minimal effort. Each plugin in Ziro is designed to integrate seamlessly with the core framework, reducing manual coding and enabling rapid development.

   For example, when a developer wants to add an authentication system or a content management interface, they can install a Ziro plugin that handles all related routes, controllers (loaders/actions), and UI components. This system's modularity allows developers to customize these features through a single configuration file, drastically reducing the time spent on setup and maintenance.

2. **Type Safety and Predictability**: A major pain point with many current frameworks is the difficulty of maintaining type safety across various parts of the application. Ziro addresses this with a type-safe system that extends throughout its routing and data-handling mechanisms. By leveraging TypeScript, Ziro ensures that data passed between components is consistently validated, reducing the risk of runtime errors and making code more predictable. This helps developers build complex applications confidently, knowing that type mismatches will be caught during development rather than after deployment.

3. **Comprehensive Configuration Management**: In frameworks like Laravel, configuration management is centralized, allowing developers to control application behavior from a single point. Ziro adopts a similar approach, providing an easy-to-use configuration file where developers can enable or disable plugins, set default values, and manage application settings. This makes it easier to adjust core features and tailor the framework's behavior to the specific needs of a project.

4. **Sequential Data Handling and Flow**: Ziro's data flow is designed to be both predictable and sequential, ensuring that data loading and state management propagate logically from parent to child routes. This sequential approach facilitates better coordination between different application parts and simplifies managing shared state, making complex data structures easier to handle.

5. **Extensibility for Advanced Features**: One of Ziro's goals is to support the development of advanced features with minimal friction. For instance, when building an application similar to WordPress or Laravel Nova, developers may need to create custom pages or modules that fit within the broader application structure. Ziro's built-in plugin system

and flexible routing make this process straightforward, supporting extensions that behave like native parts of the framework and can extend the typesafety.

The Need for a Comprehensive Foundation

While frameworks like Next.js, Nuxt.js, Astro or Remix have gained popularity for their robust capabilities, they are often focused on providing a solid base for general-purpose applications. Ziro's focus is different; it aims to provide a comprehensive, foundational platform that facilitates building developer-friendly tools and applications directly out of the box. With Ziro, developers can focus on building features that matter without having to spend an excessive amount of time setting up repetitive, common components.

Ziro in Context with the Existing Ecosystem

When compared with other frameworks, Ziro's position stands out because it bridges the gap between advanced, feature-rich platforms like Laravel and the modern JavaScript ecosystem. While tools like Astro offer efficient rendering and optimized performance, their modular approach can sometimes leave developers piecing together components manually. Ziro's approach is more holistic—it combines powerful, built-in features with the ability to enhance the framework with additional modules without additional complexity.

The modular and type-safe nature of Ziro opens up new opportunities for web developers who seek the versatility of modern JavaScript frameworks while enjoying the convenience and extensibility of established CMS platforms. By enabling developers to build feature-rich applications quickly, Ziro sets a new standard for frameworks in the JavaScript ecosystem.

3.8 Conclusion

The landscape of web development has seen remarkable evolution, moving from server-rendered content to dynamic client-side applications and now to the hybrid models supported by modern frameworks. Frameworks like Next.js, Nuxt.js, Remix and Astro have paved the way, each addressing different aspects of performance, SEO, and developer experience. However, as the demand for comprehensive, feature-rich, and scalable applications grows, so too does the need for a framework that combines the flexibility of modern JavaScript tools with the extensibility and ease-of-use found in mature CMS platforms like WordPress and Laravel Nova.

Ziro emerges as a compelling solution in this context. With its type-safe architecture, modular plugin system, and centralized configuration management, it promises to streamline the development of complex applications while promoting rapid deployment and ease of maintenance. Ziro's approach to sequential data handling, combined with its ability to integrate advanced features seamlessly, positions it uniquely within the ecosystem.

As development teams navigate a choice among existing frameworks, understanding their strengths and limitations is key. While established players like Next.js offer reliability and a well-trodden path, newer frameworks such as Astro showcase the potential for innovative, minimalistic approaches. Ziro, however, sets itself apart by catering to developers who need both foundational structure and modular extendibility. It fills the gaps left by existing solutions, providing a practical, developer-centric platform that fosters rapid application development without sacrificing flexibility or depth.

In conclusion, Ziro represents the next step in a modern developer's toolkit—an inclusive framework that blends the best of JavaScript with the tried-and-tested practices of established web development platforms. Its unique features support developers in building powerful, scalable applications that meet the demands of today's web environment, while remaining adaptable to future technological shifts. As the landscape continues to evolve, frameworks like Ziro will be at the forefront, inspiring a new era of development that is both practical and innovative.

4 Methodology

4.1 Building the Core of Ziro

The development of Ziro started with a focus on its most important part—the router. This main component, called `ziro/router`, forms the base of the framework and takes care of handling routes and interacting with browser events. The initial step of creating a solid and reliable router was crucial for setting the groundwork for the rest of the framework, ensuring a strong and consistent foundation for future development.

4.2 Creating `ziro/router`

The `ziro/router` package was designed to be as independent as possible from specific frameworks and was built entirely in JavaScript to ensure compatibility with different environments and libraries. Its main goal is to offer an API that can handle browser events smoothly, such as updating the route when a user clicks on a link or adjusting the router state when they use the back and forward buttons on the browser. This independence also ensured that the package could be reused and extended for different UI libraries like React, Vue, etc, making Ziro versatile and adaptable.

4.3 Using Test-Driven Development

The development of `ziro/router` began with a focus on rigorous testing, although it did not follow a formal test-driven development (TDD) approach. Without a fully operational environment to test the router's performance in action, Vitest was used as a critical tool for writing and running tests. These tests helped verify that the code functioned correctly and covered edge cases, contributing to a more robust codebase and preventing potential issues. Vitest, known for its speed and ease of use in JavaScript and TypeScript projects, played an essential role in ensuring the reliability of the router's functionality and streamlining the verification process. The use of Vitest, a fast and user-friendly testing tool for JavaScript and TypeScript, contributed greatly to the reliability of the codebase and streamlined the process of verifying the router's functionality.

This rigorous testing approach was especially crucial because `ziro/router` was initially developed as a client-only router. Achieving comprehensive code coverage was a top priority, given the absence of a user interface (UI) for visual confirmation of features. Consequently, testing became the primary method for

verifying the robustness and dependability of the router's functionality, ensuring it could handle real-world scenarios reliably. This part will be moved to the 'Theses' section to support the idea of Ziro's reliability and commitment to high-quality development.

4.4 Moving to `ziro/react`

After establishing a stable `ziro/router`, the next step was developing a version for React applications, called `ziro/react`. This transition allowed developers to integrate the router seamlessly within React projects, providing the benefits of Ziro's routing capabilities while maintaining the ease of use that React developers expect.

4.5 Workflow for Development

The project was structured as a monorepo, meaning that the `ziro/react` package could share dependencies and directly use the development version of `ziro/router`. This monorepo approach not only made it easier to manage dependencies but also facilitated synchronized development across packages. Any changes made to `ziro/router` could be tested and updated alongside `ziro/react`, ensuring consistency and reducing the risk of compatibility issues.

While building `ziro/react`, new needs arose that required adjustments to the `ziro/router` core. This back-and-forth process of refining both packages together was crucial for the evolution of Ziro. It ensured that enhancements and new APIs were added in a way that supported the overall goal of creating a cohesive and powerful framework.

The in-depth discussion about the benefits of using a monorepo will be moved to the 'Technical Specifications' section to provide a more detailed look at this technical setup.

4.6 Improving Type Safety with TypeScript

A detailed explanation of the `ziro/generator` package and how it worked with Vite is better suited for the 'Technical Specifications' section. This package was developed to create type-safe routes by automatically generating TypeScript types, which greatly improved code safety and developer confidence.

4.7 Adding Server-Side Compatibility

Once `ziro/router`, `ziro/react`, and `ziro/generator` were working smoothly on the client side, the next challenge was adding server-side compatibility. This involved integrating the client-side router with a server-side version to facilitate data sharing between the two. This feature was crucial for enabling Ziro to support both server-side rendering (SSR) and client-side rendering (CSR) within a unified framework. This integration allowed developers to build applications that could leverage the strengths of both rendering types, providing a seamless experience and better performance for end-users.

4.8 Improving and Refining the Framework

The process of developing `ziro/router`, followed by `ziro/react`, `ziro/generator`, and adding server-side support, was carried out in a step-by-step manner. Each phase played an important role in refining the framework and ensuring its core components remained strong, flexible, and aligned with the overall goals. The iterative development approach meant that as new challenges arose, the framework could be adapted to meet them, ensuring long-term robustness and scalability.

4.9 Results of the Development Process

This careful and step-by-step approach led to the development of Ziro's main packages with:

- Stability: The use of rigorous testing and TDD created a dependable codebase that can be trusted for reliable performance.
- Flexibility: The `ziro/router` package was built to be adaptable and can be used across different projects and environments, not just React.
- Scalability: The modular architecture allows for easy additions of new features and plugins, making it straightforward to expand Ziro's capabilities.
- Type Safety: The integration of TypeScript and automated type generation minimized errors and boosted developer confidence in building and maintaining applications.

5 Theses

6 Technical Specifications

7 Results

8 Future Works