# An agile architecture that does not bleed [1]

## 1 Introduction

An agile architecture is a structure for designing an application in a flexible and iterative manner where the goal ”(...) is to minimize the human resources required to build and maintain the required system.” (Martin, 2017, p. 16). A key aspect of this approach is maintaining a clean architecture that is easy to build upon, a point emphasized repeatedly by Martin. However, in practice, this proved to be more challenging than expected. Working within a team unfamiliar with the technology and under constant time pressure often forced us to opt for quicker, less clean solutions that had to be iterated upon later, to be able to meet our delivery deadlines. Simultaneously, we recognized the importance of selecting a technology stack that not only facilitated rapid development but also resulted in a scalable, maintainable, and performant product. These experiences led me to the formulation of the central problem statement: How can the strategic use of established technologies contribute to the development of an agile architecture, while simultaneously ensuring the delivery of a high-quality product for the customer?

## 2 Not reinventing the wheel

One of the most beneficial decisions we made at the start of the project was to establish the architecture and implement cross-cutting features such as authentication and dark-mode immediately. This approach facilitated incremental development, as there were no major architectural changes required before developing individual features. We achieved this by adhering closely to the documentation for React, a popular library we utilized, which prescribes its own modular architecture. We opted for React as our primary technology due to its robust ecosystem and widespread adoption. As per the 2022 State of JavaScript survey, React is utilized by over 80% of JavaScript developers (State of JS, 2022).

Unlike the well-known Model-View-Controller (MVC) architecture (Gamma, Helm, Johnson, & Vlissides, 1994, p. 14), React employs a component-based architecture. The application is divided into smaller, reusable pieces (components) that manage their own state and rendering. In this context, the state of the component can be seen as the ”model”, the components themselves as the ”view”, and user interactions with the component as the ”controller”.

Adopting the architecture provided by React offered numerous advantages since it facilitates the Don’t Repeat Yourself (DRY) principle. This enabled us to reuse components in multiple places which enhanced readability and made development faster. However, React is merely a library and lacks several features necessary for a Create, Read, Update, and Delete (CRUD) application, for instance routing between pages and data retrieval.

While these features could be implemented by creating our own backend API and add packages that solve specific problems like routing, we opted not to ”reinvent the wheel”. Instead, we chose to use Next.js, the framework recommended in the React Documentation (React Team, 2024). Next.js integrates many cutting-edge features for React, transforming it into a full-stack framework. This eliminated the need for a separate backend codebase, saving much time since now we could retrieve data directly within the components, and creating a new route was as simple as creating a new folder. It also significantly improved performance, as the React package bundle is never sent to the frontend when using Next.js.

Reflecting on our experience, using established solutions like React and Next.js expedited feature development and eased infrastructure concerns, despite our time constraints and limited familiarity. The tight coupling of backend and frontend that comes with Next.js made it more challenging for team members who preferred to focus solely on one aspect, but also simplifed the architecture. We learned that mastering React before tackling Next.js is best to avoid being overwhelmed.

## 3 Know where to bleed

One might wonder if it would be simpler to use a Backend As A Service (BAAS) like Firebase instead of Next.js, thereby eliminating the need to manage our own database. Indeed, using Firebase or a similar service could simplify development

[1] Declaration about the use of artificial intelligence: ”No use of artificial intelligence.”

and speed up the process significantly. However, this approach comes with its own set of trade-offs, particularly when considering the principles of an agile architecture.

As former Twitch Developer and Entrepreneur Theo Browne points out, when adopting new or proprietary technologies, it is crucial to understand the associated costs and challenges (Browne, 2023). Browne uses the phrase ”know where to bleed” to encapsulate this idea. In the context of our project, choosing to use a service like Firebase could lead to ”bleeding” in terms of scalability and maintainability. Firebase may offer rapid development, but it can also lead to high costs as the application scales and can make it difficult to migrate to another service in the future. As Martin puts it: ”To be effective, a software system must be deployable. The higher the cost of deployment, the less useful the system is.” (Martin, 2017, p. 112).

Instead, we decided to stick with a more traditional approach for our database management, using SQL. SQL has been a reliable choice for businesses since the 1970s, offering scalability and flexibility in terms of hosting options. We chose PostgreSQL for our database, which we could interact with directly within our React components, thanks to Next.js.

This decision also aligned well with our team’s learning goals, as many members were concurrently learning SQL while developing the application. However, managing our own database and interacting with it through SQL does introduce additional complexity. While this might seem overkill for our application, I believe that adhering to professional standards and best practices is crucial for maintaining an agile architecture.

In conclusion, knowing ”where to bleed” means understanding where it makes sense to innovate and where it’s better to stick with tried-and-true solutions. In our case, we chose to innovate with our use of Next.js and React, while sticking with the reliable and scalable SQL for our database management. For the customer, having a cost-effective solution that allows for seamless migration between providers is also a great benefit.

## 4 Reflections and Recommendations

Reflecting on the project, the journey was filled with learning experiences and challenges. The decision to use React and Next.js, while initially daunting due to the steep learning curve, proved to be beneficial in the long run. It allowed us to focus on feature development and provided a scalable and maintainable architecture. I believe the key isn’t the choice of framework, but its correct implementation. Consequently, when opting for a framework, prioritizing proper codebase architecture is crucial.

For future projects I would want to use a simpler technology stack if the team is not experienced. An architecture, no matter how agile, loses its effectiveness if it becomes a learning hurdle for team members, hindering their ability to contribute effectively. Therefore, balancing agility with the team’s proficiency could lead to a more productive and truly agile environment. As Martin wrote, ”A software system that is hard to develop is not likely to have a long and healthy lifetime. So the architecture of a system should make that system easy to develop, for the team(s) who develop it” (Martin, 2017, p. 111).

Based on my experiences, for future teams embarking on a similar project, I would recommend the following to establish an agile architecture:

1. **Invest time in learning:** Before diving into the project, invest time in understanding the technologies you plan to use. In our case, understanding React before moving on to Next.js would have made the learning process less overwhelming.
2. **Leverage established technologies:** Don’t reinvent the wheel. Use established technologies and frameworks that facilitate rapid development and provide a scalable and maintainable architecture.
3. **Know where to bleed:** Understand the trade-offs associated with the technologies you use. It’s crucial to know where it makes sense to innovate and where it’s better to stick with tried-and-true solutions.
4. **Maintain a clean architecture:** Even under time pressure, strive to follow best practices and establish a clean architecture for the technologies you use. This will make it easier to build upon your project and ensure its long-term maintainability.

## 5 Conclusion

In conclusion, the process of creating an agile architecture while ensuring the quality of the product for the customer is a delicate balancing act. It involves making informed decisions about which technologies to use, understanding the trade-offs associated with these decisions, and maintaining a clean architecture despite time pressures.

Strategic use of established technologies, such as React, Next.js, and SQL, was key. These technologies facilitated an agile architecture and ensured product quality. In essence, my experiences highlight the importance of strategic technology choices. They showed that with the right strategies, it's possible to develop an agile architecture and deliver a high-quality product.

## References

Browne, T. (2023, July 7). *Bleeding edge**. `https://youtu.be/uEx9sZvTwuI`. (Accessed: 2024-04-05)

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

React Team. (2024). *Start a new react project*. `https://react.dev/learn/start-a-new-react-project`. (Accessed: 2024-04-05)

State of JS. (2022). *Front-end frameworks*. `https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/`. (Accessed: 2024-04-05)