From JavaScript to Elixir: Lessons from Reimplementing an Industrial Home Care System

Ricardo de Oliveira Ferreira
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
ricardo.o.ferreira@ufv.br

Lucas Vegi
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
lucas.vegi@ufv.br

Abstract

This paper presents an industrial case study evaluating the migration of a real-world healthcare SaaS application from a JavaScript stack to Elixir. We reimplemented 144 API endpoints and representative frontend screens while preserving the original business logic, and compared both implementations through end-to-end, stress, and microbenchmark experiments. The results show that Vue.js frontend achieved lower end-to-end latency, whereas the Elixir/Phoenix backend consistently outperformed the equivalent Node.js implementation in throughput and scalability, reaching up to 2,096× higher throughput. Overall, the results suggest that migrating the backend to Elixir can substantially improve scalability while reducing cloud infrastructure costs. These findings provide quantitative evidence to support technology migration decisions.

Keywords

Elixir, JavaScript, Benchmarking, Software Migration

1 Introduction

The company MedYes[1] develops and maintains a Software-as-a-Service (SaaS) platform for home healthcare management, named CareYes. The platform enables managers to efficiently oversee employees, patients, inventory, and financial operations through a unified web-based system. Healthcare professionals and caregivers can also access up-to-date patient information, enabling them to understand each patient's needs better and provide more personalized, human-centered care.

The production version of CareYes is entirely based on a JavaScript stack, comprising a Vue.js frontend and a Node.js backend. As the platform has grown, the company has experienced scalability challenges, particularly performance degradation during periods of high demand. Although the system has undergone several code refactoring efforts to improve resource utilization, its current implementation still requires substantial computational resources to sustain increasing workloads. Consequently, the company's cloud infrastructure costs have increased by approximately 13% per month. Because the platform is commercialized under a subscription-based licensing model, these growing operational expenses directly reduce its profit margins.

To investigate a more cost-effective alternative, this work evaluates a potential technological migration of the system to Elixir.[2] We partially reimplemented the application using the Phoenix framework,[3] a mature ecosystem for developing scalable, fault-tolerant, and high-performance web applications [7], while preserving the original business logic. We then compared the original JavaScript implementation and its Elixir counterpart through end-to-end, stress, and microbenchmark experiments.

Elixir was selected because of its intrinsic characteristics, particularly its concurrency model and its ability to efficiently leverage multicore processors, enabling highly available systems while requiring fewer computational resources [5]. Furthermore, successful industrial migrations to Elixir, including Pinterest [6] and Bleacher Report [7], motivated its adoption in this study. In both cases, migrating from Java, Python, and Ruby-based systems to Elixir reduced the number of servers required to support the same workload by tens of times, leading to lower infrastructure costs while maintaining or improving system performance.

Our contributions are threefold: (i) an empirical comparison of equivalent JavaScript and Elixir implementations of an industrial SaaS application; (ii) evidence that Vue.js provides better end-to-end performance, whereas Elixir/Phoenix achieves higher backend throughput and scalability; and (iii) quantitative evidence that migrating the backend to Elixir could substantially reduce operational infrastructure costs in the analyzed industrial context.

The remainder of this paper is organized as follows. Section 2 presents the research question and methodology adopted in this study. Section 3 reports and discusses our results. Finally, Section 4 concludes the paper and outlines directions for future work.

2 Methodology

The main objective of this study is to evaluate whether migrating a JavaScript-based web application to Elixir can improve performance while reducing computational resource consumption, thereby providing quantitative evidence to support technology migration decisions in industrial settings. Accordingly, we formulated the following research question:

- **RQ:** How do equivalent JavaScript and Elixir implementations compare in performance and resource utilization?

To answer this research question, the study was organized into three stages. First, we **analyzed** the company's source code to understand the system architecture, technical design decisions, and business rules. Next, we partially **reimplemented** the system in Elixir using the Phoenix framework while preserving its original behavior. Finally, we **evaluated** both implementations by benchmarking four representative endpoints. Figure 1 summarizes the steps we followed in our methods. Next, we also detail these steps.

[1] https://medyes.com.br/
[2] https://elixir-lang.org/
[3] https://www.phoenixframework.org/

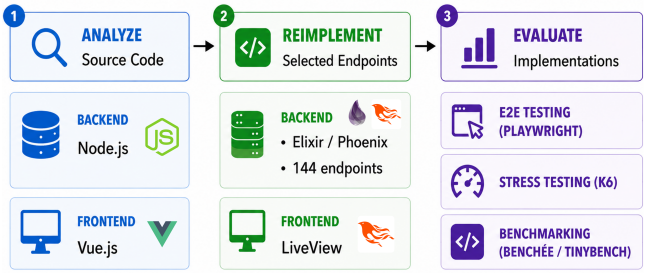

Figure 1: Overview of our methodology.

1) Code analysis: We manually analyzed CareYes, implemented with a Node.js backend (approximately 950 source files and 1,200 REST API endpoints) and a Vue.js frontend (approximately 1,600 source files spanning 150 screens), to understand its architecture, business rules, and coding patterns before reimplementation. The backend follows an MVC architecture, while the frontend is organized into reusable Vue single-file components.

2) Code reimplementation: We reimplemented 144 backend REST API endpoints using Elixir and the Phoenix Framework [7], preserving the original business logic to ensure that the experiments reflected only the impact of the underlying technology. Whenever possible, the implementation followed the architectural and behavioral decisions of the legacy system, while adopting Phoenix-specific practices when they did not affect the system behavior. The selected endpoints correspond to those consumed by the company's mobile application, providing a representative subset of the production system. The frontend was partially reimplemented using Phoenix LiveView, a server-driven UI technology that enables interactive web applications with minimal client-side JavaScript [10]. We selected representative screens covering the application's core workflows, including patient listing, patient calendar, clinical records, visits, photo upload, and gallery management.

3) Benchmarking: We evaluated both implementations using three complementary benchmarking strategies. First, we performed end-to-end (E2E) tests with Playwright [8] to measure the complete execution flow from the user's perspective, including browser rendering, network communication, and backend processing. Second, we conducted stress-testing experiments with K6 [2] to evaluate throughput and scalability under increasing levels of concurrent requests. Finally, we performed microbenchmarks using Benchee [11] (Phoenix) and Tinybench [9] (Node.js) to measure isolated backend performance, eliminating network and frontend overhead.

To ensure a fair comparison, both implementations were evaluated under equivalent configurations. Playwright executed the same test suite in three independent, sequential runs without retries using Google Chrome. Following benchmarking guidelines [1, 3, 4], the reported results correspond to the arithmetic mean of these executions. Both frontends ran locally and accessed their respective backends deployed on AWS. Low standard deviations across all experiments indicate stable execution conditions. K6 adopted identical linear ramp-up profiles, increasing the workload by 200 virtual users (VUs) every two minutes until reaching the target concurrency, followed by a sustained peak period and a gradual ramp-down. Both implementations used the same MySQL database configured with a connection pool of 45 connections. Benchee and Tinybench were configured with 10 s of warm-up and 30 s of measurement time. Minor framework-specific adaptations were required to account for architectural differences between Phoenix and Node.js while preserving the behavior of the evaluated operations.

All experiments were executed on an AWS EC2 `r6i.xlarge` instance equipped with a 4-core Intel Xeon Platinum 8375C processor (2.90 GHz) and 30.91 GB of RAM. Four representative operations were evaluated across all experiments: *Login*, *List Patients*, *New Evolution*, and *Add Photo*. These operations were selected because they represent authentication, data retrieval, data creation, and file upload workloads, respectively. The complete benchmarking scripts and configurations are available in our replication package.

3 Results

Table 1 compares the end-to-end execution times of the Phoenix and Vue implementations. The Vue implementation outperformed Phoenix for all evaluated endpoints, with average execution times ranging from 14.2% to 115.3% lower. One possible explanation for the observed differences lies in the architectural distinctions between the two implementations. Unlike Vue, which renders the user interface on the client through conventional HTTP requests, Phoenix LiveView maintains a persistent WebSocket connection and performs part of the rendering and state management on the server [10]. These additional processing and communication steps may introduce extra latency in end-to-end interactions, although the present study does not isolate their individual impact.

Table 1: Playwright end-to-end execution times.

Endpoint	Phoenix	Vue	Overhead
Login	9.92 s	5.36 s	85.1%
List Patients	10.55 s	4.90 s	115.3%
New Evolution	15.14 s	10.20 s	48.4%
Add Photo	15.11 s	13.23 s	14.2%

Table 2 presents the microbenchmark results obtained with Benchee for Phoenix and Tinybench for Node.js. Performance is reported as *iterations per second* (IPS), a throughput metric where higher values indicate better performance [3]. The *Speedup* column reports the ratio between Phoenix and Node.js IPS, with values above 1× favoring Phoenix and below 1× favoring Node.js.

Overall, Phoenix achieved higher throughput in three of the four evaluated endpoints. It outperformed Node.js by 1.24× for *Login* and showed nearly identical performance for *Add Photo* (1.04×). The most significant result was observed for *New Evolution*, where Phoenix achieved approximately 2,096× the throughput of Node.js. Although this remarkable difference suggests that this workload particularly benefits from Phoenix's execution model, the architectural factors responsible for this behavior were not isolated and therefore require further investigation. Node.js outperformed Phoenix only for *List Patients*, achieving approximately 1.52× higher throughput.

Memory results should be interpreted with caution because Benchee measures memory allocated by a BEAM process [5], whereas Tinybench reports Node.js heap usage. Despite these differences,

Table 2: Throughput and memory comparison between Phoenix and Node.js using Benchee and Tinybench.

	Throughput (IPS)			Memory (KB)		
Endpoint	**Phoenix**	**Node.js**	**Speedup**	**Phoenix**	**Node.js**	**Diff.**
Login	14.91	12.00	1.24×	184.08	71.67	+156.8%
List Patients	280.91	427.00	0.66×	99.03	220.61	-55.1%
New Evolution	901.51	0.43	2096.5×	83.48	2017.01	-95.9%
Add Photo	4.15	4.00	1.04×	596.77	277.69	+114.9%

Table 3: Stress-testing results obtained with K6.

	Latency (Avg.)			Throughput (Requests)			Execution	
Endpoint	Phoenix	Node.js	Speedup	Phoenix	Node.js	Diff.	Success (Phoenix/Node)	VUs
Login	1.83 s	11.70 s	6.39×	105.9k	23.7k	4.47×	98.0/99.9%	200
List Patients	768 ms	1.91 s	2.49×	678.4k	412.6k	1.64×	100/100%	800
New Evolution	34 s	59 s	1.74×	2.6k	1.5k	1.73×	99.4/0.5%	60
Add Photo	494 ms	684 ms	1.38×	2.0k	1.8k	1.11×	100/100%	6

Phoenix consumed less memory for *List Patients* and *New Evolution*, but more for *Login* and *Add Photo*. Overall, Phoenix achieved higher computational throughput than the equivalent Node.js implementation, with memory consumption depending on the evaluated workload. These results complement the end-to-end experiments by confirming Phoenix's performance advantages in isolated execution scenarios, where external factors such as network latency and browser rendering are eliminated.

Table 3 summarizes the stress-testing results obtained with K6. Overall, Phoenix consistently sustained higher request throughput and lower average response times than the equivalent Node.js implementation across all evaluated endpoints. The largest improvements were observed for the *Login* and *List Patients*, where Phoenix processed approximately 4.47× and 1.64× more requests, respectively, while maintaining substantially lower average response times.

For the *Login* endpoint, Phoenix completed requests in an average of 1.83 s, whereas Node.js required 11.7 s. Despite both implementations being limited by password hashing and the database connection pool, Phoenix maintained a success rate of 98%, indicating stable behavior under the evaluated workload.

Similar behavior was observed for the *List Patients*. Using the same workload of 800 virtual users, Phoenix processed more than 678k requests, approximately 1.64× more than Node.js, while reducing the average response time from 1.91 s to 768 ms. Since this endpoint is database-intensive, the results suggest that Phoenix utilized the available resources more efficiently, despite the database connection pool and CPU cores remaining the primary bottlenecks.

The most notable result was observed for *New Evolution* endpoint, where Phoenix maintained a 99.4% success rate, compared with only 0.5% for Node.js. This suggests that Phoenix remained stable under a workload that saturated the Node.js implementation

Overall, the stress-testing experiments reinforce the results obtained from the microbenchmarks. While the end-to-end experiments favored the Vue-based application in terms of isolated user interaction latency, the K6 results demonstrate that Phoenix scales more effectively under high concurrency, sustaining higher throughput and higher request completion rates under heavy workloads.

To illustrate the potential cost savings of a backend migration, we extrapolated the stress-testing results for the *Login* endpoint. On the evaluated AWS EC2 `r6i.xlarge` instance, the Elixir/Phoenix implementation sustained approximately 59 requests/s, compared with 13 requests/s for Node.js. Assuming a target workload of 1,000 requests/s and a hypothetical infrastructure cost of US$1 per server-hour, 17 Elixir servers and 77 Node.js servers would be required. This corresponds to an estimated monthly infrastructure cost of US$12,240 for Elixir versus US$55,440 for Node.js (24 hours/day, 30 days/month), representing an approximate 78% cost reduction. This example is illustrative only and should not be interpreted as a production cost estimate.

4 Conclusion and Future Work

This paper presented an industrial case study comparing equivalent JavaScript and Elixir implementations of a healthcare SaaS application. Although Vue.js achieved lower end-to-end latency, Phoenix delivered higher backend throughput and scalability than Node.js. These findings suggest that completing the backend migration could substantially reduce cloud infrastructure costs. Future work includes replicating this study in other industrial systems and deployment environments.

Artifact Availability

The complete benchmarking results and automation scripts are available at https://doi.org/10.5281/zenodo.21248993. The source code of the JavaScript and Elixir implementations is proprietary and cannot be made publicly available.

Acknowledgements

This research was supported by CNPq (PIBIC grant, 2025–2026) and MedYes: https://medyes.com.br/.

References

[1] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 57–76.

[2] Grafana Labs. 2017. K6 Open Source. https://k6.io/open-source/.

[3] Brendan Gregg. 2020. *Systems Performance: Enterprise and the Cloud* (2 ed.). Addison-Wesley Professional.

[4] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons.

[5] Saša Jurić. 2024. *Elixir in action* (3 ed.). Manning.

[6] Ben Marx, José Valim, and Bruce Tate. 2018. *Adopting Elixir: From Concept to Production* (1 ed.). The Pragmatic Bookshelf.

[7] Chris McCord, Bruce Tate, and José Valim. 2019. *Programming Phoenix 1.4: Productive |> Reliable |> Fast* (1 ed.). Pragmatic Bookshelf.

[8] Microsoft. 2020. Playwright Documentation. https://playwright.dev/.

[9] Mohammad Bagher. 2022. Tinybench Documentation. https://tinylibs.github.io/tinybench/.

[10] Bruce Tate and Sophie DeBenedetto. 2025. *Programming Phoenix LiveView: Interactive Elixir Web Programming Without Writing Any JavaScript* (1 ed.). Pragmatic Bookshelf.

[11] Tobias Pfeiffer. 2016. Benchee Documentation. https://benchee.hexdocs.pm/.