information

Review

Revisioning Healthcare Interoperability System for ABI Architectures: Introspection and Improvements

João Guedes [1], Júlio Duarte [2,*], Tiago Guimarães [2] and Manuel Filipe Santos [2]

[1] Information Systems Department, University of Minho, Azurém Campus, 4800-058 Guimarães, Portugal; a89237@alunos.uminho.pt
[2] Algoritmi Researsh Center, University of Minho, Azurém Campus, 4800-058 Guimarães, Portugal; tsg@dsi.uminho.pt (T.G.); mfs@dsi.uminho.pt (M.F.S.)
* Correspondence: julio.duarte@algoritmi.uminho.pt; Tel.: +351-253510319

Abstract: The integration of systems for Adaptive Business Intelligence (ABI) in the healthcare industry has the potential to revolutionize and reform the way organizations approach data analysis and decision-making. By providing real-time actionable insights and enabling organizations to continuously adapt and evolve, ABI has the potential to drive better outcomes, reduce costs, and improve the overall quality of patient care. The ABI Interoperability System was designed to facilitate the usage and integration of ABI systems in healthcare environments through interoperability resources like Health Level 7 (HL7) or Fast Healthcare Interoperability Resources (FHIR). The present article briefly describes both versions of this software, learning about their differences and improvements, and how they affect the solution. The changes introduced in the new version of the system will tackle code quality with automated tests, development workflow, and developer experience, with the introduction of Continuous Integration and Delivery pipelines in the development workflow, new support for the FHIR pattern, and address a few security concerns about the architecture. The second revision of the system features a more refined, modern, and secure architecture and has proven to be more performant and efficient than its predecessor. As it stands, the Interoperability System poses a significant step forward toward interoperability and ease of integration in the healthcare ecosystem.

Keywords: ABI; healthcare; continuous integration; continuous deployment; software; development; testing; HL7; FHIR; interoperability

Citation: Guedes, J.; Duarte, J.; Guimarães, T.; Santos, M.F. Revisioning Healthcare Interoperability System for ABI Architectures: Introspection and Improvements. *Information* **2024**, *15*, 745. https://doi.org/10.3390/info15120745

Academic Editor: Yutong Xie

Received: 16 October 2024
Accepted: 9 November 2024
Published: 21 November 2024

1. Introduction

The integration of Adaptive Business Intelligence (ABI) systems [1] in healthcare is a widely discussed topic, not only due to their ability to manage the vast amounts of data generated daily, but also because of the anticipated benefits of integrating an evolving intelligent system into enhancing societal care. Furthermore, these systems also support and help hospital administrators make strategic decisions that are integral to healthcare organization operations [2,3].

To expedite the path to transparency and interoperability of these solutions, the scientific community is conducting numerous studies that aim to ensure the development of technological architectures that are adequately equipped to meet these requirements [4,5] while focusing on widespread implementation and high precision [6]. A critical aspect of adopting this type of technology is its interoperability with other systems. The ability of ABI systems to interoperate with other separate systems within the healthcare ecosystem enables two-way communication, further enhancing their efficiency and effectiveness.

ABI systems, despite being highly effective and promising, can be very hard to integrate and a very frustrating roadblock, resulting in creating more complexity and reducing efficiency in their target organization.

There are a few key factors that can make these systems harder to work with and adopt, which are born from the nature of the machine learning models themselves and the state of the ABI systems:

- There are no well-defined guidelines or standards when it comes to ABI's machine learning model's availability and documentation.
- The complex nature of machine learning models can make them very hard and brittle when integrated into any environment, while maintaining data fidelity.
- Current ABI implementation depends on the target's environment systems, and technical incompatibilities are almost guaranteed, making this process very challenging and error-prone.
- The integration of the individual models of an ABI system requires different and unique strategies that often depend uniquely on the developers of the model or system.
- ABI integration depends on its models and the target environment's systems' ability to communicate and interchange data. This leads to entirely new layers and logic being developed unnecessarily just to interface with different models. This responsibility is usually up to the integration team, which often builds lots of different solutions that increase the brittleness and complexity of the systems.

One of the biggest pushes and focused developments toward this goal was an Interoperability System designed to be an enhancing layer for the ABI architecture. This new layer would leverage interoperability resources, such as HL7, to ensure seamless communication and integration of ABI systems in the healthcare ecosystem in a flexible and controllable fashion. However, as with any software, ongoing refinement is necessary to address existing flaws and ensure relevance [7].

The system was redesigned and rebuilt with the goal of addressing the problems and inefficiencies of the first design into a second, more robust, and streamlined revision.

This paper presents a comparative analysis between the original and revised versions of the ABI Interoperability system. The first iteration, while functional, revealed several inefficiencies that prompted the need for a more streamlined and robust design. The second version, completed recently, incorporates significant improvements, including enhancements in code quality through automated testing, better development of workflows via CI/CD pipelines, and integration of the FHIR pattern. The revised system also addresses the security concerns raised by the earlier architecture, offering a more secure and stable solution.

The aim of this paper is to examine these changes and their impact on the overall ABI development ecosystem. The key questions this analysis will address are: How do architectural refinements contribute to the system's robustness? What role does the integration of the FHIR pattern play in improving interoperability? How do new security features mitigate the risks previously encountered in the system? Through this exploration, the paper seeks to provide insights into the future of ABI system interoperability within the healthcare sector.

2. Analysis and Comparison Methods

This study follows a comparative design aimed at evaluating two iterations of the ABI Interoperability System [7]. Version 1 refers to the first design of the system, whereas Version 2 represents a redesign to address the shortcomings of the first implementation.

This study focuses on four key aspects:

- System architecture and workflow improvements

- Interoperability improvements
- Security improvements
- System's performance

During this assessment, a combination of qualitative and quantitative metrics was used due to the nature of the available data. Both architectures were qualitatively analyzed, and workflow improvements were based on the literature on modern software development patterns and trends. The system's performance was tested through load tests using the tool JMeter to perform tests in different scenarios simulating real-world traffic loads [8]. These tests also helped in testing the interoperability robustness of the solution using various interoperability datasets. The code security was measured via SaST [8] scanning with the Checkmarx SaST tool combined with manual result validation for False Positive reduction.

3. Analyzing the First Revision of the Interoperability System

The Interoperability System is a software solution that allows for seamless integration and usage of ABI systems through healthcare interoperability resources like HL7 or FHIR resources [7]. This system was developed as another layer for ABI architectures [1] that allows and facilitates communication with machine learning models or any other system through message queue services, such as RabbitMQ or ActiveMQ [7].

It was based on a microservices architecture and was developed as a modular solution in which all its components can be iterated separately, which promotes their independence in both development and execution [7,9,10].

3.1. Software Architecture and Design

Figure 1 illustrates the conceived technological architecture for the Interoperability System, which spans three distinct layers: technology, service, and presentation.

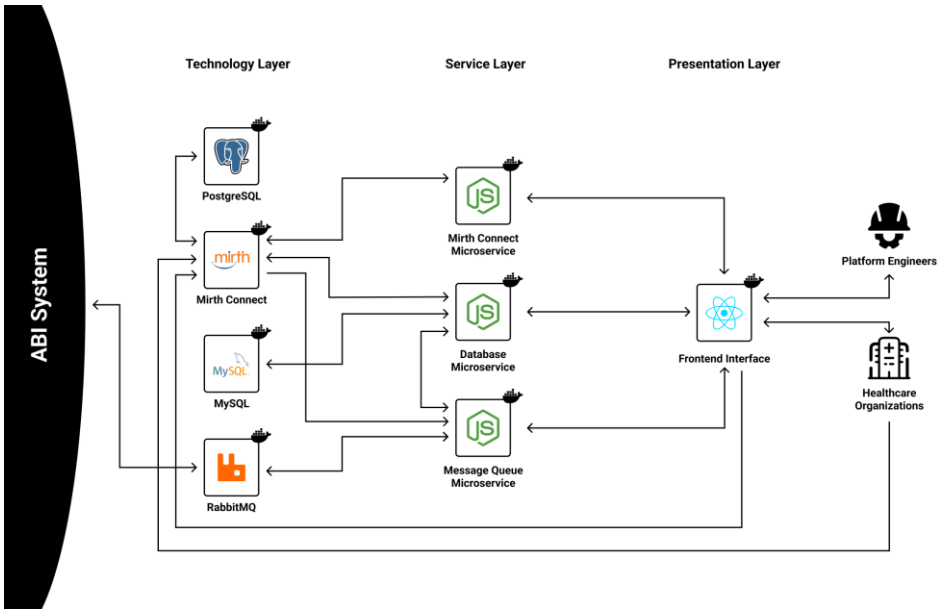

Figure 1. Interoperability System Technological Architecture.

The Technology layer consists of the backbone of the entire system and architecture, as it includes all software solutions that undergird or support it [7]. It is made up of a database for storing all the system's data, Mirth Connect (and PosgreSQL) for the robust handling of HL7 resources [11], and a message queue solution for communication with the outside environment.

The service layer is where all the system's business logic lies. The components in this layer are the ones that will carry out the core functionality of the system [7]. These services are responsible for and capable of interfacing with the technology layer to access and manipulate data or trigger diverse system functions, as well as interacting with each other to execute complex operations. Designed to be modular and reusable, it promotes efficiency and consistency throughout the system and is made up of three main services: the Mirth Connect service, interoperability service, and message queue service [6]. Each of these services is responsible for interfacing with a different component of the technology layer and contains all the secrets and logic to do so. It is also notable that every service in this layer is directly accessible to any component in the system.

The presentation layer acts as a bridge between a complex system and its users [6]. It provides a user-friendly way of interacting with all services that make up the system. It is composed of a single web application that can communicate with the service layer to accomplish every action or function present in the system, not only abstracting its users of its complexity, but also providing a very convenient and easy way to do so.

The division of responsibilities among the different layers allows for a high degree of adaptability and scalability since each component is a singular and independent piece of software that must collaborate with the other components to achieve the system's objectives. This way, changes in one component can be implemented and will not cause disruption to the other layers.

This layered architecture is a common pattern in software design, as it helps manage complexity by breaking the system down into smaller parts that can be independently iterated on and tested, enhancing the system's quality, stability, and maintainability [7,9,10,12].

Although the service layer was developed to be able to support lots of different technological choices and solutions, the system currently only supports MySQL as the database, Mirth Connect for HL7 handling, and RabbitMQ for message queuing [6].

3.2. Solution Software Design Patterns

At its core, the Interoperability System's main functionality is to seamlessly integrate and use healthcare interoperability resources such as HL7 or FHIR with an ABI system, more specifically its machine learning models, while the rest of its functionalities only exist to help support and fully fledge this goal [6].

As shown in Figure 2, the system's main operation can take a raw input in the form of an unaltered interoperability resource and gradually transform it into a correct input for an ABI system's machine learning model. This is accomplished via three well-defined and distinct processes that simulate the first crucial steps of the CRISP-DM methodology of data gathering/understanding and data cleanup/preparation [7,13]. These stages will work together with various user-defined moving parts to not only maximize customization at runtime, but also to ensure that the machine learning models will be used accordingly to their development.

This logic and flow design follows and adapts the popular and well-known Pipeline design pattern [14]. This pattern works through a series of operators that work sequentially, where the output of one operator serves as the input of the next, which results in a complete dependency between them. This means that if any operator fails, the flow will stop in a controlled manner as it is designed to do so [15].

As a direct response to the volatile and highly changeable nature of any machine learning model's requirements, the operators needed to implement a mechanism that would allow for the dynamic execution of the same algorithm with different content and logic. Alluding to the relationship between a game console and a video game, this behavior can be described as the Strategy design pattern where a family of algorithms is defined and encapsulated in such a way as to become interchangeable [15]. This pattern allows the content of an algorithm to change according to the situation at runtime in-

stead of the logic having to be present in the source code, making it a lot more flexible and reusable [15].

Referencing Figure 2 once more, the data flow will start with an interoperability resource from which the necessary data will be extracted, then validated to ensure data quality, and then processed to comply with how the models were developed, all according to the specified machine learning model.

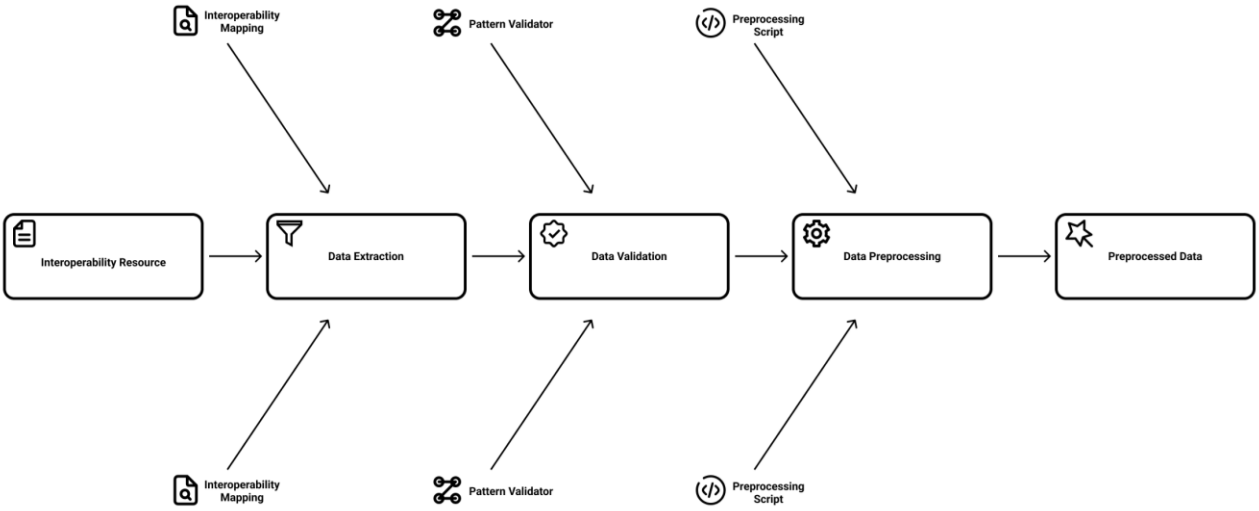

Figure 2. Interoperability system main workflow design.

3.3. System Structure

The Interoperability System was developed following the MVC (Model, View, Controller) pattern, which is one of the most known and used patterns in the industry and, as the name suggests, is composed of three logical components: Model, View, and Controller, which help the complexity of software solutions [16].

The Model component is responsible for storing and managing all data of the solution, and thus, the interoperability service fills this role as it is capable of interfacing with the technology layer.

The View component is responsible for presenting data to the final users, and it is accomplished by the web application in the presentation layer.

The Controller component is responsible for handling and executing user interactions or actions while updating the other layers accordingly. This function was assumed by all microservices in the service layer, as each one controls a different but integral part of the system.

All service layers were developed via Node.js, and all microservices followed a well-defined template complete with Docker/Docker Compose support, organized file structure, and API documentation with Swagger. In this way, all microservices can be developed in a consistent fashion following the best practices of microservice development.

This standardization of the development of the system's microservices will ensure their consistency and stability and will also allow it to take advantage of the containerization of its services at runtime [17].

The web application that makes up the presentation layer and is responsible for the View component of the system was developed using React.js and Ant Design and follows the best practices of front-end development, such as prioritizing reusable, readable, and documented code or the use of SVG files instead of PNG [18,19].

As it is, the system relies on MySQL for data storage, RabbitMQ for integration and communication with external services, and Mirth Connect as the core runtime for the solution. This runtime would consist of a single Mirth Connect channel configured to in-

terface with the rest of the system's services to perform the different stages of data processing until it was ready to be pushed via RabbitMQ.

4. Analyzing the Second Revision of the Interoperability System

The second revision of the Interoperability System consists of its most recent iteration, which builds upon the robust foundation established by the first revision and brings forth a series of enhancements, optimizations, and fixes to both the software solution and its development workflow. The next topics will describe in more detail the major and most important changes.

4.1. HL7 FHIR Support

Perhaps the most impactful change in the system's feature range was the full support of the industry standard HL7 FHIR (Fast Healthcare Interoperability Resources) pattern, making the system capable of handling FHIR resources instead of only HL7 in both JSON and XML formats [20].

This change mainly impacted the data extraction aspect of the system and influenced the entire solution because every aspect of the system needed to be redesigned to accommodate different types of interoperability resources. An entire custom engine must be developed to handle and query the FHIR specification guidelines and properly extract data from these resources [20,21].

The support for FHIR resources originated from the ever-growing popularity and degree of preference over traditional HL7 that FHIR has been seeing over recent years, becoming increasingly relevant with each passing day [20]. This can be observed in Figure 3.

At the observed rate of growth, the support for the FHIR pattern became a sudden and highly important new requirement for the second iteration of the Interoperability System.

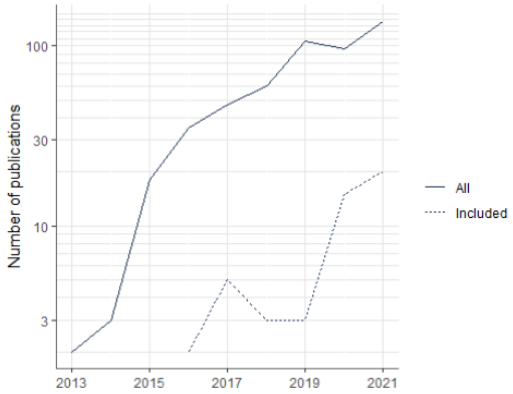

Figure 3. Published studies on FHIR over time [20].

4.2. Secure Strategy for Data Preprocessing

By far, the biggest flaw and security risk regarding the first revision of the Interoperability System was how it handled the dynamic execution of the models' preprocessors. The system accomplishes this by having a preprocessor engine that can take a previously made preprocessor and run it against the target data, resulting in preprocessed data, as shown in Figure 4.

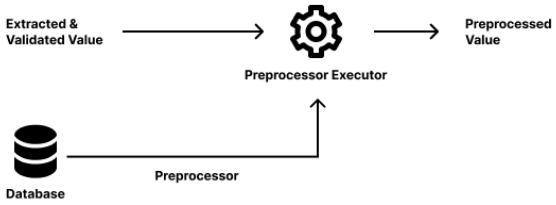

Figure 4. Preprocessing mechanism concept.

This concept worked by storing pre-written scripts (written in JavaScript) in the database to later fetch them as they were needed and executing them via JavaScript's expression evaluation method of eval [22]. This dynamic code execution technique is dangerous given its high permission privileges and potential for code injection, which can be disastrous for any software solution [23,24]. Furthermore, a technique like this implemented directly on the server will always constitute a severe security vulnerability and an extremely irresponsible practice, and thus, one of the most widely used and well-recognized mitigation strategies is to run the dynamic code off the server.

Figure 5 shows the new approach to the preprocessing mechanism, which utilizes a sandboxed environment to execute all dynamic codes separately and in isolation from the server [25]. This will guarantee the safety and security of the resources and the system itself from possible malicious code injections [25]. The preprocessor execution engine itself was rewritten to accommodate this new sandbox technique and to properly capture all errors that may arise, making this feature much more robust.

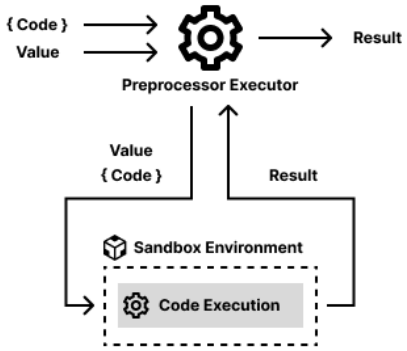

Figure 5. Preprocessing mechanism concept using sandboxing.

Apart from the introduction of sandboxing, a whitelist strategy [26] was also implemented to supplement and fortify the security of the preprocessing feature, because it only allows users to write preprocessors that contain specific and previously reviewed tokens, invalidating all others. This is much more efficient and secure than a blacklist because instead of having to ban certain terms and keywords from scripts, we must only allow for those that we are certain are not capable of being used for malicious purposes. This whitelist was implemented in a way that allows for constant iteration and update as it is not static.

4.3. Business Logic Transition from Mirth to Node.js

The first revision of the system only supported HL7 resources [27,28], and thus, the entire business logic for the dynamic extraction of data from them, validation, and preprocessing could be developed using the Mirth Connect [11,29]. This was fine and worked well enough, but the FHIR support addition forced a big paradigm change that resulted in the rewriting of the entire business logic regarding the interoperability data flow in Node.js, while only using Mirth for small HL7 data extractions. For the most

part, the system's functionality remained the same; it was the way it was executed and called that changed, providing a faster and much more realized runtime environment [30–32].

As Figure 6 illustrates, the Interoperability System's architecture did not change drastically, as it was still composed of the same layers and technologies, but the whole paradigm of operations changed completely. The layer service is now centralized in the Interoperability microservice, which consists of the main point of contact with the system. All interactions with any of the system's functions occur through this interoperability service, as it controls all incoming traffic. The main workflow was moved from Mirth Connect to Interoperability microservice, which serves only as an HL7 parser. FHIR was supported in this version; therefore, the interoperability service makes use of the FHIR schema to perform all FHIR operations. The HL7 schema was also constructed locally and all HL7 V2 operations passed through it.

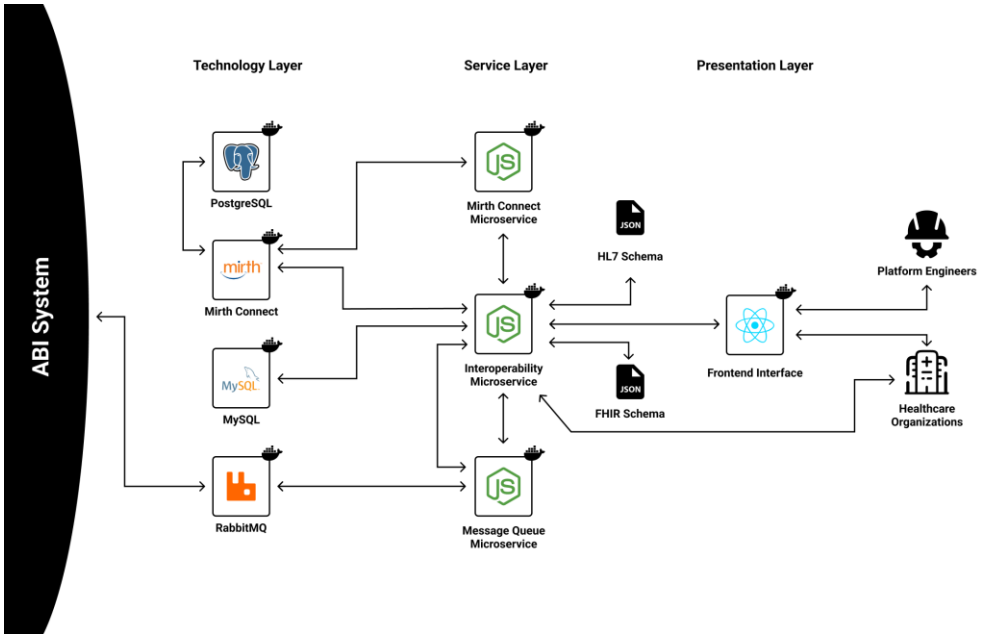

Figure 6. Interoperability System Revision 2 Technological Architecture.

4.4. Version Control

The first revision of the system was developed with the help of a version control system (Git), but it did not benefit much from it besides having some rudimentary commit history and safe storage in a Git repository manager (Github) [33]. For single and slow development, this strategy was not a problem, but for managing different versions and introducing collaboration to the various repositories, it needed to be improved.

The first step toward this improvement would be the establishment of a well-defined branching strategy that could ensure the consistency and quality of all the work being performed and delivered [34,35].

The flow represented in Figure 7 demonstrates the branching strategy adopted to handle all future developments for all separate repositories that make up the system. There are two long-term branches, dev and master, for testing changes and hosting the final version of the code, respectively, as well as short-term branches for the development of new features and introduction of changes.

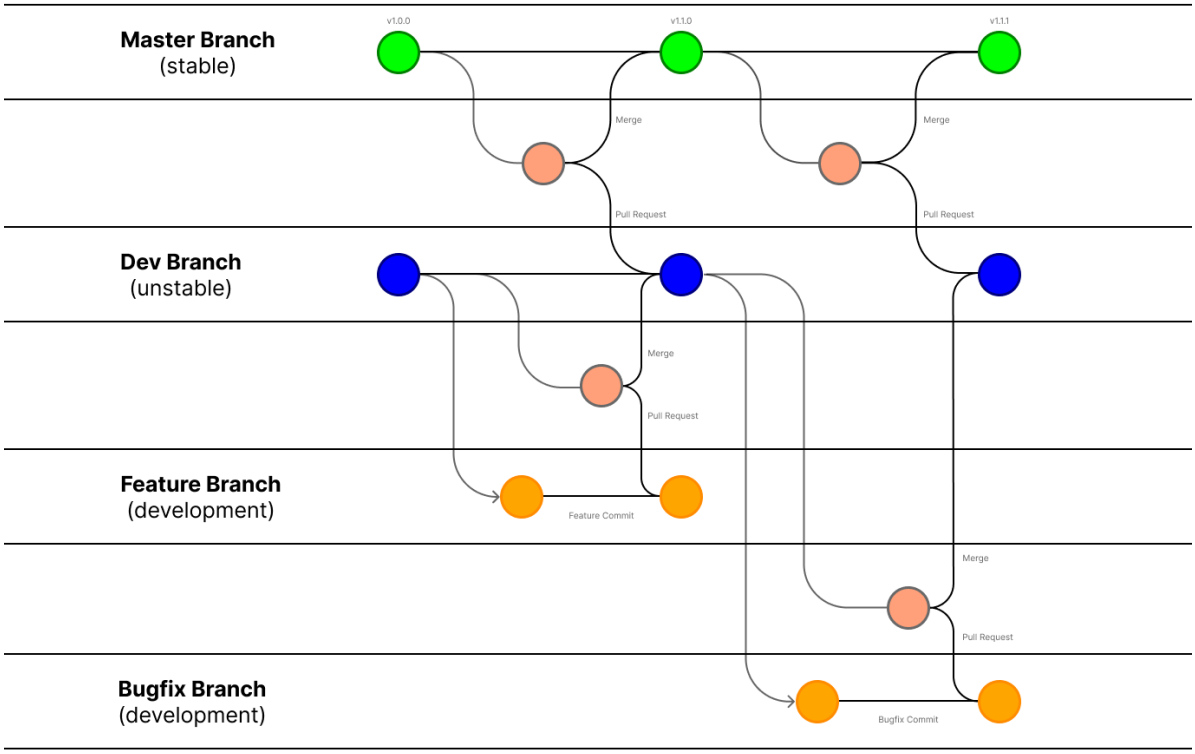

Figure 7. Interoperability system repository branching strategy.

This branching strategy includes the following three tiers of branches:

- Master Branch—This branch includes the production code, which consists of the most recent stable version of the project. It is only iterated via pull requests from the tested changes in the development branch. Usually, changes to this branch result in an increase in product version.
- Dev Branch—Contains all testing code and features that still need to be verified. It is expected that a large part of the features in this branch are unstable and, therefore, not yet ready to move on to the master branch.
- Feature Branches—The iteration-based development of the repositories relies on the creation of these feature branches, which are very short-term and highly focused versions of the product that are meant to develop new features or fix problems. Once the changes are completed, they are merged into the development branch for testing, and the feature branch must be deleted.

These branches were then correctly configured to ensure that the long-term branches could not be directly updated only through pull requests, and these required code review from the code owners before being merged with the target branches.

As expected, the master branch will include many iterations during its lifecycle, which can become very confusing due to the uncertainty of the number of changes between each version. This uncertainty typically results in software breaking or failure due to version conflicts [36,37]. This problem introduces the second step toward improving the development of the system, which is standardizing the different versions of the production code that have been released. Semantic Versioning (SemVer) was the standard adopted to this end, as it provides a way to communicate instantly what kinds of changes a new version holds [36,37].

It is composed of three significant numbers separated by dots, with increments in the first number representing the introduction of breaking changes, the second representing the introduction of new features, and the last representing the correction of previous problems.

The same concept was applied to the commit messages in the repositories, as they can also become very confusing and meaningless. The pattern of Conventional Commits was applied and allowed for a clear understanding of the changes being made, but also for some level of automation [38]. It structures commit messages with a type, optional scope, and description. Types include 'fix', 'feat', and others, each correlating with a change type in Semantic Versioning. The 'BREAKING CHANGE' type indicates a major change. This structure aids in creating an explicit commit history and facilitates automated tooling.

4.5. Continuous Integration and Deployment

Automation is the heart of the modern software development ecosystem as it can ensure that tests are performed, quality standards are met, repetitive work is always conducted consistently, and it frees developers and engineers so that they can focus on more important tasks [39].

The automation strategy implemented in the Interoperability System's repositories had the following goals:

- Automatic test execution to ensure consistency.
- Automatic deployment in a test environment so that the entire system can be tested.
- Automatic release of tagged versions and their well-documented changes.

These goals were accomplished via GitHub Actions, which allow for automation based on repository changes [40,41].

The implemented CI/CD strategy is shown in Figure 8, which shows that it relies on the creation of feature branches to introduce changes in the code. These changes must be merged into the development branch, but only after being approved by the code owners and the success of the automatic unit tests.

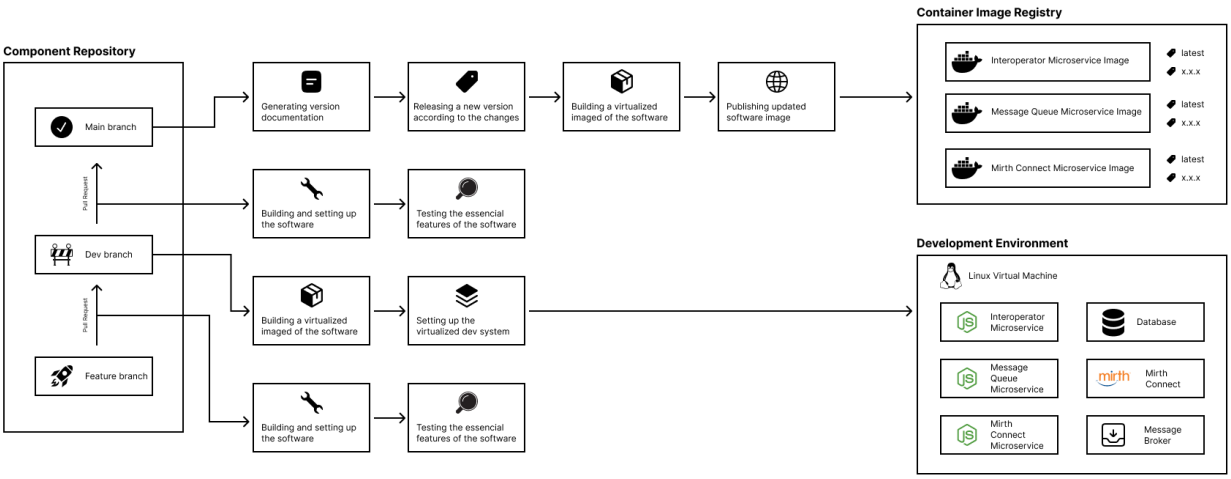

Figure 8. CI/CD Automation Strategy.

Once the changes are merged with the development branch, they are automatically deployed in a testing environment hosted on a separate Linux virtual machine via an actions runner so that the changes can be tested in a production-like environment with end-to-end testing. This testing environment will feature all the necessary system components, only updating the immediate changes in a specific component.

After end-to-end tests are performed, either manually or automatically, the changes must be merged from the development branch to the master branch, which also needs approval by the code owners to run automatic unit tests.

When the changes enter the master branch, their documentation is automatically generated based on the structured commit messages, a new version tag is created ac-

cording to SemVer, and a new image of the software is created and published to the GitHub Container Registry [42].

4.6. Unit Testing

Unit testing is a critical part of software development, as it ensures that the individual units of code, such as functions or modules, are working as expected and are not broken from iteration to iteration [43,44].

Regarding the microservices, unit tests were performed with the help of the JavaScript testing framework Jest, as it is very popular and simple to use [45]. Jest features were used to write simple tests for each relevant function concisely and mock the system database in CRUD-related tests (Create, Read, Update, and Delete).

For the web application, Cypress was used to test each React component individually using its very powerful API to not only traverse the entirety of the volume of components but also to mock the requests made to the server.

These test workflows were developed and implemented in the CI/CD automation workflow to ensure that no changes could break the features or components currently being tested.

4.7. End-to-End Testing

End-to-end testing is a methodology based on the concept of black box testing, in which the flow of an application or system is tested from start to finish, simulating the product's usage in production [46].

In systems like the Interoperability System, which is composed of many moving parts and components, these kinds of tests are essential to guarantee their stability and functionality. For this reason, end-to-end tests were created and are a very important part of the current CI/CD strategy, as they run on the development environment to ensure that changes in the dev branch will not break functionality [46].

Once again, the tool of choice was Cypress, given its versatility, power, and ease of use, and it was seamlessly integrated into the CI/CD process via GitHub Actions to ensure the correct behavior of the entire system from the final user's perspective [46]. To this end, 12 distinct end-to-end tests were conducted to validate all the system's functionalities.

5. Comparing Both Revisions of the Interoperability System

Both versions of the Interoperability System were individually analyzed via the respective source code and documentation, and they show distinct differences in architecture, features, and overall performance. While the first revision is focused on laying the foundation for the interoperability workflow and a functional system, the changes made for the second revision are much more focused on improving the design and development workflow while mitigating the flaws found.

5.1. System Architecture and Workflow

The first revision of the system was designed based on a microservice architecture [9,10] and distributed different functions and responsibilities between singular and isolated services, with the different components separated into well-defined layers. In this way, each technology component would have a dedicated service to interface with, and these services would be free to communicate with each other or be called via their respective APIs [30]. This approach relied heavily on Mirth Connect as an entry point for the processing workflow, as a single channel would receive a request and pass it along to the rest of the services via their APIs until, eventually, the processed result would reach the RabbitMQ queues where it would be ready to be consumed by the rest of the ABI system.

For the second revision, this architecture largely remained the same when it came to the distribution of layers and responsibility, but the paradigm of communication between the services shifted toward an API Gateway pattern [10], meaning that all traffic would flow through an entry service and then be propagated to all components in the system. The interoperability microservice became the API Gateway of the system and would interface with the rest of the services via their respective API, whereas the communication channels for these services were restricted only to the gateway service. The business logic was also moved to this microservice rather than running in a difficult-to-manage Mirth Channel. This meant that Mirth Connect would only be used as an HL7 parser instead of being the core of data processing.

The support for the interoperability resources HL7 and FHIR was improved by localizing all schema lookups to locally defined schemas rather than querying public resources, mitigating another possible point of failure.

The rest of the changes were made to the development workflow of the Interoperability System in the following form:

- Implementation of a branching strategy based on long-term branches with strict guidelines on merging iterations from short-term branches [34,35]
- Integration of Semantic Versioning on software releases for all components following the SemVer pattern [36,37]
- Enforcing Conventional Commits pattern for commit structure [38]
- Continuous integration and deployment for automated linting, testing, and version release [39]
- Automated unit tests for code quality and reliability using Jest [47,48]
- End-to-end testing in dedicated development server and deployment [46]
- Software virtualization streamlined with Docker [49,50]

These changes contribute to code quality, maintainability, and reliability as the components become more stable and documented [36,37], thanks to all the testing policies imposed on development. The branching strategy allowed for well-structured projects and iterations while promoting collaboration between multiple sources [34,35]. The changes in documentation culture and versioning allow for better observability and ease of use following the common practices of software release. Lastly, the automated streamlining of the services' releases as virtualized images improves the consistency of the solution and simplifies the process of integration.

5.2. Interoperability Support

The first revision completely focused on HL7 V2 [51], as it used Mirth Connect as the core for all logic regarding data processing. This limited the system to Mirth's functionality and capability of HL7 handling. This meant that the system's interoperability functionality could only evolve at the same rate as the Mirth Connect. Thus, the first revision could only process and handle HL7 resources. The second revision changed this by radically changing the core of the system to a dedicated service that could route requests to Mirth if they regarded HL7 V2 or elsewhere if they required any other interoperability engine. The main focus of the second revision was to support FHIR resources [20]. To achieve this, an entirely custom engine was built from scratch that, similar to Mirth, could parse FHIR messages and extract any necessary data. The engine was built using the FHIR R5 and R4B specifications, but the support was implemented in a way that allowed for the future support of more versions.

5.3. Security and Performance

Security was a major concern regarding the first revision of the Interoperability System, given that it was not its focus. The system was still designed to minimize risks, but further security analysis is necessary to properly assess and mitigate them [7]. One of the most problematic security flaws in the system was the dynamic code execution regard-

ing the controllers [7,22,52] since any code present in the controllers would be executed "no questions asked" in the preprocessing stage. This was properly mitigated using a combination of sandboxing and token whitelisting techniques [23,25]. This ensured that the dynamic code that was going to be executed did so in an isolated Node.js instance, and only a few selected tokens were allowed to be present in the executed expressions.

The localization of the interoperability resource schemas also contributed to the isolation of the services, which no longer required them to access the Internet for queries related to them.

The restriction and proper configuration of the request origin [53] between the services also reduced the communication to critical services, like the Message Queue and Mirth Connect, to the bare minimum. In this context, the only reachable service for external communication was the interoperability service, which was the system's gateway.

6. Comparative Tests Results

While the features and architectural changes can be compared and analyzed qualitatively as better or worse, the system's performance and security can be measured quantitatively by executing the system, performing load tests [8], and security scans. In this fashion, quantitative comparative tests relied on Apache JMeter for load testing [54], Docker statistics for hardware usage, and the Checkmarx SaST tool for static vulnerability analysis [55]. As expected, the second revision yielded better results for all the metrics measured. A more detailed report on the tests performed can be found in the following subsections.

6.1. Load Testing

Load testing was conducted to evaluate and assess the system performance under simulated real-world conditions and loads [8]. This test measured how well the two systems handled large volumes of data and traffic, simulating the loads expected in the healthcare field.

To perform this test, both versions of the system were deployed on two different Linux virtual machines running the same configuration and hosted by the same system. This was performed to make sure that any differences in the results were a direct result of the improvements made. Only HL7 resource datasets were used during the load tests because it is the only message type supported by both systems.

Similar to the performance tests performed on a Healthcare Hub Server [56], the load test used Apache JMeter [54] and spanned three different load scenarios simulating 100.000 requests distributed by 200, 1000, and 2000 concurrent users over five minutes. This test yielded results representing low, normal, and high levels of traffic to assess which revision was better able to handle, in general, the loads expected in real-world scenarios.

As Figure 9 indicates, there is an expected trend of increasing average response times as the number of concurrent requests increases. This was expected because of the load increase, but it is also apparent that the second revision is, on average, faster than the first revision. This is surely because of the business logic transition from Mirth Connect to Node.js and the optimization of the service's communication. This means that if the services were to be migrated to a more performant runtime, such as golang [57], the average response time would be able to go even lower. The lowest all-time response time was 50 ms, with 200 concurrent requests per second. The highest response time was from the first revision, with 2000 concurrent requests. Analyzing the results, we can assess that the second revision is ~72%, 50%, and 20% faster than the first revision in all scenarios. This trend shows that as the load increases, the gap between both systems shortens.

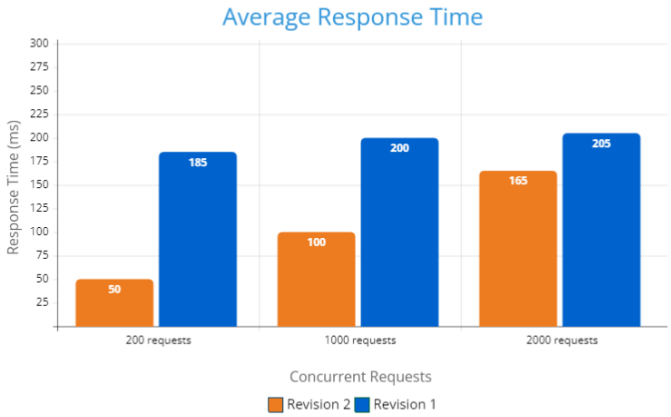

Figure 9. Average Response Time Performance Test.

Looking at the results of the throughput comparison in Figure 10, we can verify that at lower loads, both versions can process the requests and respond successfully; however, as loads intensify, the number of requests that can be processed decreases significantly. In other words, as the average time increased, the throughput decreased. This is to be expected from these tests [56], but comparatively, the second revision of the system is better at handling higher loads while keeping a higher throughput rate of ~20%, ~36%, and 31% higher in low, normal, and high loads, respectively. Another aspect to note is that, similar to the average response time, as loads increase, the gap first jumps very high and then seems to become shorter, indicating that there will be a load level where the performance might be about the same.

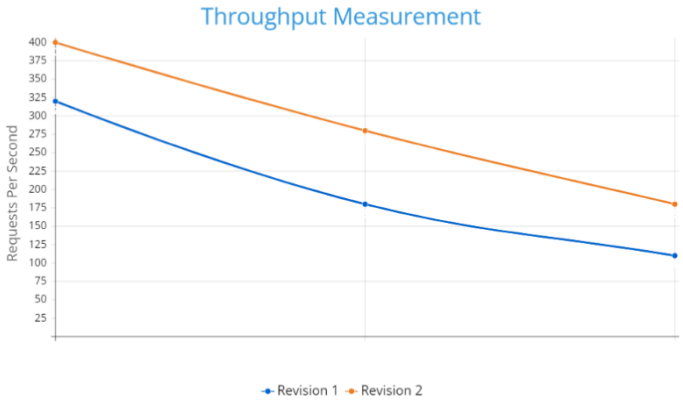

Figure 10. Throughput Performance Test.

Using the statistics features bundled with Docker; it was possible to record the usage of resources over time [58], of which the CPU and Memory percentages were the metrics of interest. Only the peak values for these metrics were considered for comparison.

As shown in Figure 11, both systems reached maximum CPU usage and were close to maximum Memory usage. This indicates that the performance of resource usage is not significantly different between the two revisions at high loads. However, for lower and more manageable loads, there is a clear difference between the two revisions, where the second revision is much more efficient while processing all requests. This is the case because the Mirth Connect can be very resource intensive, while Node.js is a much smaller and lighter runtime. The API Gateway pattern also reduced the number of internal requests and API calls, and thus reduced resource usage. This reduction in resource usage

was expected, reaching an all-time high of ~56% more CPU efficient and ~20% more memory efficient.

Both systems eventually reach the maximum resource usage, and this might happen because of the relatively low resources given to both VMs and/or because there is still a lot of room for optimization.

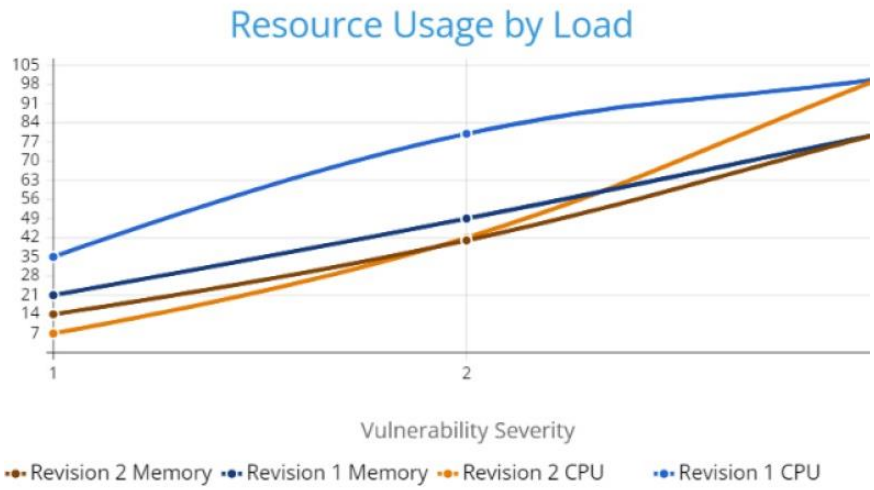

Figure 11. CPU/Memory Usage by load.

6.2. Security Testing

To assess the security of the solution, an SaST [55] scan and evaluation was conducted using Checkmarx SaST. This type of scan breaks the source code into tokens, and then, using known vulnerabilities, checks the flow of data to search for patterns and behaviors known to cause dangerous software vulnerabilities. By nature, these scans tend to be quite heavy regarding False Positive results, and thus, analyzing each one is necessary to maximize the insight taken from that scan.

In this fashion, the Interoperability service source code was chosen to be scanned because it is the most crucial service in the system, resulting in 343 vulnerabilities for the first revision and 290 for the second one. These results are assumed to be quite exaggerated by False Positives, so a manual analysis of the results was necessary, resulting in only 116 actual exploitable results for the first revision and 77 for the second.

Checkmarx SaST considers four types of vulnerabilities and separates them by increasing levels of severity as Information, Low, Medium, and High, with Information and Low, indicating the least concerning results and High indicating the most concerning. Figure 12 illustrates the differences in the frequency of True Positive results found in both source codes for each revision. It also indicates that between the first and second revisions, there was a large decrease in High results and small decreases in Low and Medium results. It is also notable that there were no increases in the results, so no new vulnerabilities were introduced.

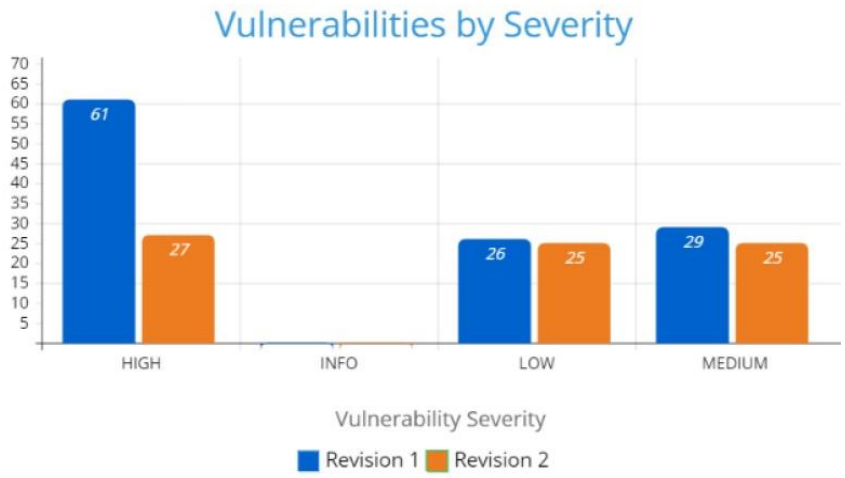

Figure 12. True Positive vulnerabilities by severity.

Detailing a little more about what vulnerabilities are present in the source codes of both revisions, Figure 13 shows the frequency by vulnerability.

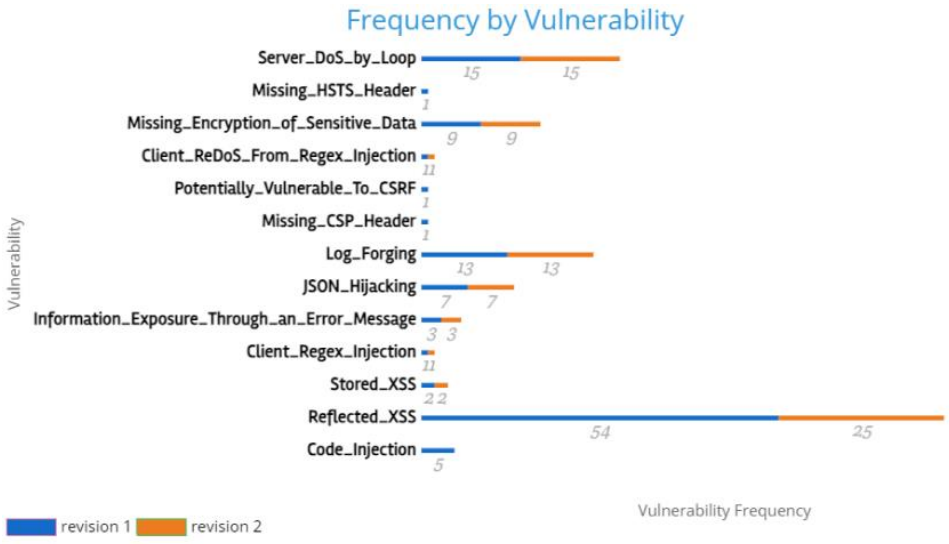

Figure 13. Frequency by vulnerability.

As Figure 13 indicates, most of the vulnerabilities remain between both revisions, but there are a few that were reduced or completely removed, as represented in Figure 12. These vulnerabilities were mostly High severity, and their existence posed a big risk to the security of the solution. Simple fixes, such as preventing CSRF and HSTS or CSP headers, were also implemented due to their nature as an insecure configuration or missing request header [59,60].

The elimination of code injection vulnerabilities [59] is by far the most important improvement. This was likely because of the dynamic execution of the code implemented for preprocessing. This was mitigated by the combination of a sandbox and whitelist strategy [23,25,61], as previously mentioned. The reflected XSS vulnerability also decreased by a large margin owing to slight improvements in the validation of the user input.

The second revision reflected a big improvement in the number of exploitable high severity vulnerabilities present in the system, but there are still quite a few that need mitigation. As for Medium and Low, their numbers did not change significantly.

6.3. Interoperability Testing

One of the main features of the second revision of the Interoperability System is the support of the FHIR schema, which allows the processing of FHIR resources. Because it is a separate engine, it will not be conditioned by Mirth Connect, and it is potentially faster since it is much smaller. The same load tests were performed only with the second revision with FHIR resources, and the results are represented in both Figures 14 and 15.

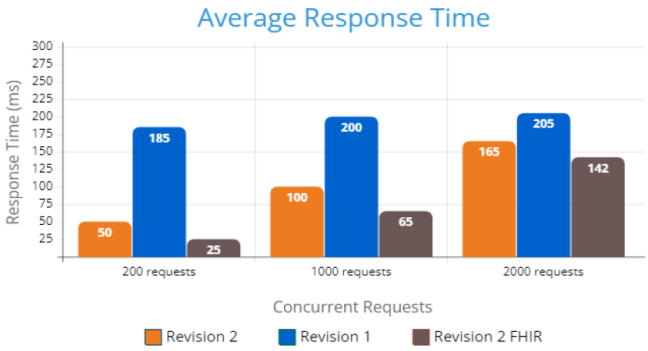

Figure 14. Average Response Time Performance Test with FHIR.

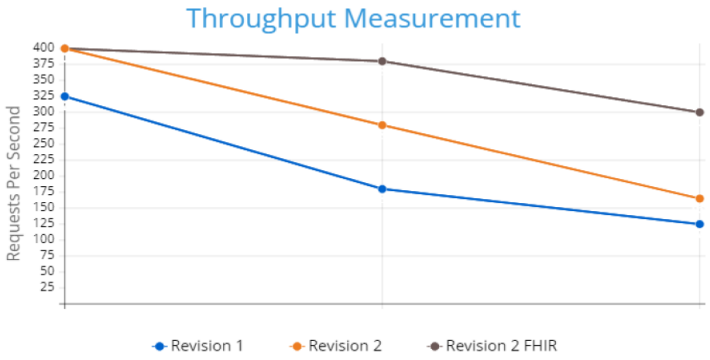

Figure 15. Throughput Measurement Test with FHIR.

The average response times seem to be much lower with FHIR resources in all testing scenarios, although it also ramps up with 2000 concurrent requests. This is largely due to the FHIR engine being written entirely from scratch and being relatively simpler and lightweight compared to Mirth Connect. The results seem to indicate that Mirth constitutes a bottleneck when it comes to the speed of processing. The throughput also becomes much healthier and only starts to decrease with 2000 concurrent requests.

7. Discussion

The second revision of the ABI Interoperability System represents a significant improvement over the previous version in terms of performance, security, design, and interoperability. Redesigning the system around the transition of business logic from Mirth Connect to a dedicated service allowed for a much more focused, refined, and secure architecture. This also allowed following the API Gateway microservice design pattern [10].

The various improvements to the development workflow contributed to a cleaner, organized, and well-documented code base, which was reflected in the significant decrease in vulnerabilities found during security vulnerability scans.

The core of the system now runs on a dedicated Node.js microservice, which allows for significantly shorter processing times, higher throughput, and lower resource usage. This indicated that the system became more efficient because of the runtime change,

which means that it could become even more efficient if the services were to be rewritten in a more performant language, such as golang or rust [57].

The support for FHIR resources was a very important addition, as it aligned the system with current healthcare data standards, addressing the interoperability limitations present in the first revision [20]. This is consistent with the growing demand for FHIR-based systems in the modern healthcare environment [20,21].

In comparison to the processing of HL7 resources, the processing of FHIR resources through the system yielded much better results in terms of performance, which might indicate a possible bottleneck in the current Mirth Connect implementation. This might constitute the grounds for building a similar engine for parsing HL7 messages.

The most impactful security improvement was the implementation of sandboxing [25] and whitelisting of the preprocessors' dynamic execution in order to prevent the code injection vulnerability [59]. This mitigation was confirmed, along with other smaller vulnerabilities, using the Checkmarx SaST tool [8]. Taking a closer look at the vulnerabilities pointed out by the SaST tool, 77 other vulnerabilities are confirmed to still be present, of which 52 are somewhat concerning as they represent High to Medium severity vulnerabilities.

8. Limitations

While the second revision of the Interoperability System introduces very significant improvements over the first one, this study is subject to several limitations that should be acknowledged. The load tests and performance metrics demonstrated were obtained from simulated healthcare environments using standard datasets. These components were designed to replicate real-world scenarios; however, they cannot fully account for the variability, complexity, and unpredictability of actual healthcare ecosystems. Other factors not present in this document can account for the decrease in the performance and reliability of the system. Further testing, preferably in live environments, would provide a much more accurate assessment of the system's scalability and reliability.

Another limitation is the system's current focus on HL7 and FHIR. This support covers the most widely adopted interoperability standards in healthcare [20,51] but does not offer any alternative for environments that rely on other or proprietary standards.

Regarding the performance assessment, this study only took into consideration the average response time of the system and request throughput. Other metrics, such as long-term scalability, could prove problematic in future studies, as the system's capability to operate with a long-term and full database was not tested. In addition, the stress of long sessions with high loads was not tested. Systems that are designed to handle vast amounts of data need to be able to operate under sustained high loads over long periods of time without performance degradation. These tests are essential for validating the system's resilience over time.

SaST tools like Checkmarx SaST are great tools for continuous and automatic code security testing, but they are quite limited and should not be used as a unique solution for security assessments due to their natural limitations [55]. These tools can identify dangerous patterns by parsing and tokenizing the source code, and they are prone to erroneous reports without vigilance. Mitigating False Positives is fairly simple, but False Negatives are harder because an SaST tool will never return results for which it does not have support. Human eyes are always necessary in conjunction with these tools, and dedicated security assessments and breakdowns are needed to properly assess the security of the system.

This study has primarily focused on the theoretical improvements, simulations, and internal testing of the Interoperability System. Further and broader validation in live environments, using real patient data, and in collaboration with healthcare IT teams is necessary to paint a real-world picture of the system's behavior and capabilities. These activities would provide valuable feedback on user experience, ease of integration, and unexpected challenges that may arise.

9. Conclusions

The second revision of the ABI Interoperability System consists of a major improvement over the previous version and the advancement of interoperability in the healthcare ecosystem. A substantial improvement toward this goal is the system's ability to process both HL7 and FHIR resource data formats and schemas, further aligning itself with current healthcare data exchange protocols and patterns [7,20,21,51]. The transition of core business logic to an isolated Node.js service led to faster processing times, higher throughput, and enhanced scalability. This change in paradigm allowed for a more modern API Gateway pattern of microservice architecture [9,10].

The introduction of secure coding practices, particularly through sandboxing [25] and the whitelist strategy for handling dynamic code execution, has addressed the critical vulnerabilities identified in the first revision. This improvement not only makes the system more secure but also reduces the risks associated with code injection, which is a major concern in healthcare systems that handle sensitive data [59].

The system's maintainability and stability have been greatly improved through the implementation of CI/CD pipelines [39] that enable continuous integration, testing, and delivery. These practices ensure that the system can be updated efficiently and safely while enabling and promoting collaboration without introducing complexity. The practices introduced in the development workflow help guarantee code quality from multiple sources through linting, stability through testing [46], documentation through conventional commit messages, and availability through Semantic Versioning [36,37].

Despite these improvements, certain areas still require attention and future iterations. The system supports FHIR and HL7, but offers no other alternative for environments incompatible with them. It is necessary to perform real-world tests to confirm the long-term scalability of the system and identify possible bottlenecks that might be harder to spot. There might be a possible bottleneck in the Mirth Connect implementation, which could justify the development of an "in-house" parser for HL7. Runtimes and technologies other than Node.js, MySQL, and RabbitMQ should be explored and tested, as they might provide better performance and stability. Lastly, SaST scanning does not provide an in-depth security assessment; further dedicated studies need to be conducted to confirm all possible security risks.

In summary, the second revision of the Interoperability System offers a much more robust, secure, scalable, modern, and functional solution than its predecessor. While there are still areas that warrant further testing and investigation, this revision lays a strong foundation for future development and classifies this system as a useful addition to the ABI ecosystem regarding interoperability in healthcare. The findings of this study might possibly fuel the next iterations of the system by analyzing all the changes made, performing comparative testing, and reporting the results in the form of possible improvements.

Author Contributions: Conceptualization, J.G. and J.D.; methodology, J.G and M.F.S.; software, J.G.; validation, J.D. and T.G.; formal analysis, J.G. and T.G.; investigation, J.G. and J.D.; resources, J.G.; data curation, J.D. and T.G.; writing—original draft preparation, J.G.; writing—review and editing, J.G., J.D., M.F.S.; visualization, T.G.; supervision, J.D. and M.F.S.; project administration, M.F.S.; funding acquisition, M.F.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by – Fundação para a Ciência e Tecnologia, within the R&D Units Project: UIDB/00319/2020.

Institutional Review Board Statement: The study was conducted under the Declaration of Helsinki and approved by the Ethics Committee of Centro Hospitalar Universitário do Porto for studies involving anonymized patients' data regarding their interaction with health professionals.

Informed Consent Statement: Not applicable (all data used in this study was completely anonymized).

Data Availability Statement: The dataset analyzed during the current study is not publicly available due to the Administrative Council of Centro Hospitalar Universitário do Porto's authorization for research purposes only and not for publication but is available from the corresponding author on reasonable request.

Acknowledgments: This work has been supported by – Fundação para a Ciência e Tecnologia, within the R&D Units Project: UIDB/00319/2020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Michalewicz, Z.; Schmidt, M.; Michalewicz, M.; Chiriac, C. *Adaptive Business Intelligence*; Springer: Berlin/Heidelberg, Germany, 2007.
2. Ashfaq, A.; Nowaczyk, S. Machine learning in healthcare-a system's perspective. *arXiv* **2019**, arXiv:1909.07370. https://doi.org/10.1145/1235.
3. Lopes, J.; Braga, J.; Santos, M.F. Adaptive Business Intelligenceplatform and itscontribution as a support in the evolution of Hospital 4.0. Procedia Computer Science. 2021. In proceedings of The 12th International Conference on Ambient Systems, Networks and Technologies Networks (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40), Warsaw, Poland, 23–26 March 2021. https://doi.org/10.1016/j.procs.2021.04.016.
4. Aceto, G.; Persico, V.; Pescapé, A. Industry 4.0 and Health: Internet of Things, Big Data, and Cloud Computing for Healthcare 4.0. *J. Ind. Inf. Integr.* **2020**; *18*. https://doi.org/10.1016/j.jii.2020.100129.
5. Tian, S.; Yang, W.; Le Grange, J.M.; Wang, P.; Huang, W.; Ye, Z. Smart healthcare: making medical care more intelligent. *Glob. Health J.* **2019**, *3*, 62–65. https://doi.org/10.1016/j.glohj.2019.07.001.
6. Wang, F.; Casalino, L.P.; Khullar, D. Deep Learning in Medicine - Promise, Progress, and Challenges. In *JAMA Internal Medicine*; American Medical Association: Chicago, IL, USA, 2019; Volume 179, pp. 293–294. https://doi.org/10.1001/jamainternmed.2018.7117.
7. Guedes, J.; Duarte, J., Manuel, M., Quintas, C., Cunha, J., Guimarães, T. and Santos, M. Interoperability Architecture proposal for Adaptive Business Intelligence Systems in Healthcare Environments. In proceedings of The 15th International Conference on Ambient Systems, Networks and Technologies Networks (ANT) / The 7th International Conference on Emerging Data and Industry 4.0 (EDI40), Hasselt, Belgium, 23–25 April 2024. https://doi.org/10.1016/j.procs.2024.06.113.
8. Jiang, Z.M.; Hassan, A.E. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Trans. Softw. Eng.* **2015**, *41*, 1091–1118. https://doi.org/10.1109/TSE.2015.2445340.
9. Di Francesco, P.; Lago, P.; Malavolta, I. Architecting with microservices: A systematic mapping study. *J. Syst. Softw.* **2019**, *150*, 77–97. https://doi.org/10.1016/j.jss.2019.01.001.
10. Taibi, D.; Lenarduzzi, V.; Pahl, C. Architectural Patterns for Microservices: a Systematic Mapping Study. *Closer* **2018**, 221–232. Available online: http://microservices.io/patterns/index.html (accessed on 6 May 2024).
11. Lin, J.; Ranslam, K.; Shi, F.; Figurski, M.; Liu, Z. Data migration from operating EMRs to OpenEMR with mirth connect. *Stud. Health Technol. Inform.* **2019**, *257*, 288–292. https://doi.org/10.3229/978-1-61499-951-5-288.
12. Zdun, U.; Queval, P.-J.; Simhandl, G.; Scandariato, R.; Chakravarty, S.; Jelic, M.; Jovanovic, A. Microservice Security Metrics for Secure Communication, Identity Management, and Observability. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–34. https://doi.org/10.1145/3532183.
13. Azevedo, A.; Santos, M.F. KDD, SEMMA AND CRISP-DM: A parallel overview. 2008. Available online: https://recipp.ipp.pt/bitstream/10400.22/136/3/KDD-CRISP-SEMMA.pdf (accessed on 5 Novenber 2023).
14. Vermeulen, A.; Beged-Dov, G.; Thompson, P. The Pipeline Design Pattern. In Proceedings of the OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems, Austin, TX, USA, 15–19 October 1995.
15. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA. 1995.
16. Potdar, A.M.; Narayan, D.G.; Kengond, S.; Mulla, M.M. Performance Evaluation of Docker Container and Virtual Machine. *Procedia Comput. Sci.* **2020**, *171*, 1419–1428. https://doi.org/10.1016/j.procs.2020.04.152.
17. Majeed, A.; Rauf, I. MVC Architecture: A Detailed Insight to the Modern Web Applications Development. *Peer Rev. J. Solar Photoenergy Syst.* **2018**, *1*, 1–7.
18. Reddy, M.P. Analysis of Component Libraries for React JS. *IARJSET* **2021**, *8*, 43–46. https://doi.org/10.17148/iarjset.2021.8607.
19. Rawat, P.; Mahajan, A.N. ReactJS: A Modern Web Development Framework. In *International Journal of Innovative Science and Research Technology*; IJISRT Digital Library: Rajasthan, India 2020, Volume 5. Available online: www.ijisrt.com (accessed on 24 May 2024).
20. Vorisek, C.N.; Lehne, M.; Klopfenstein, S.A.I.; Mayer, P.J.; Bartschke, A.; Haese, T.; Thun, S. Fast Healthcare Interoperability Resources (FHIR) for Interoperability in Health Research: Systematic Review. *JMIR Public Health Surveill.* **2022**, *10*, e35724. https://doi.org/10.2196/35724.

21. Ayaz, M.; Pasha, M.F.; Alahmadi, T.J.; Abdullah, N.N.B.; Alkahtani, H.K. Transforming Healthcare Analytics with FHIR: A Framework for Standardizing and Analyzing Clinical Data. *Healthcare* **2023**, *11*, 1729. https://doi.org/10.3390/healthcare11121729.
22. Richards, G.; Hammer, C.; Burg, B.; Vitek, J. The eval that men do: A large-scale study of the use of eval in javascript applications. In *European Conference on Object-Oriented Programming*; Springer: Berlin/Heidelberg, Germany, 2011, pp. 52–78. https://doi.org/10.1007/978-3-642-22655-7_4.
23. Staicu, C.-A.; Pradel, M.; Livshits, B.; Darmstadt, T.U.; Livshits, B. Understanding and Automatically Preventing Injection Attacks on Node.js. In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2016. Available online: https://nodesecurity.io/advisories/ (accessed on 15 May 2024).
24. Vasilakis, N.; Staicu, C.A.; Ntousakis, G.; Kallas, K.; Karel, B.; Dehon, A.; Pradel, M. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In Proceedings of the ACM Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; pp. 1821–1838. https://doi.org/10.1145/3460120.3484535.
25. Laurén, S.; Rauti, S.; Leppänen, V. A survey on application sandboxing techniques. *ACM Int. Conf. Proc. Ser. Part* **2017**, *F132086*, 141–148. https://doi.org/10.1145/3134302.3134312.
26. Cesarano, C.; Natella, R. Securing an Application Layer Gateway: An Industrial Case Study. In Proceedings of the 2024 19th European Dependable Computing Conference (EDCC), Leuven, Belgium, 8–11 April 2024. http://arxiv.org/abs/2401.05961.
27. AlQudah, A.A.; Al-Emran, M.; Shaalan, K. Medical data integration using HL7 standards for patient's early identification. *PLoS ONE* **2021**, *16*, e0262067. https://doi.org/10.1371/journal.pone.0262067.
28. Noumeir, R. Active Learning of the HL7 Medical Standard. *J. Digit. Imaging* **2019,** *32*, 354–361. https://doi.org/10.1007/s10278-018-0134-3.
29. Rodriguez, J.C.C.; Stäubert, S.; Löbe, M. Automated import of clinical data from HL7 messages into open clinica and tran SMART using mirth connect. *Stud. Health Technol. Informatics* **2017**, *228*, 317–321. https://doi.org/10.3233/978-1-61499-678-1-317.
30. Doglio, F. REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development, Second Edition. In *REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development*, 2nd ed.; Apress Media LLC: New York, NY, USA, 2018. https://doi.org/10.1007/978-1-4842-3715-1.
31. Janne, K. Designing a Node.js full stack web. 2023. Available online: https://www.theseus.fi/bitstream/handle/10024/793330/Kinnunen_Janne.pdf;jsessionid=AE5B98B0D949590ED3C35B15D668530F?sequence=2 (accessed on 28 April 2024).
32. Pereira, C.R. Building APIs with Node.js. In *Building APIs with Node.js*; Apress: New York, NY, USA, 2016. https://doi.org/10.1007/978-1-4842-2442-7.
33. Cosentino, V.; Izquierdo, J.L.C.; Cabot, J. A Systematic Mapping Study of Software Development with GitHub. *IEEE Access* **2017**, *5*, 7173–7192. https://doi.org/10.1109/ACCESS.2017.2682323.
34. Emad, S.; Christian, B.; Thomas, Z. The Effect of Branching Strategies on Software Quality. In Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering And Measurement, Lund, Sweden, 19–20 September 2012.
35. Phillips, S.; Sillito, J.; Walker, R. Branching and Merging: An Investigation into Current Version Control Practices. In Proceedings of the 4th international workshop on cooperative and human aspects of software engineering, Honolulu, HI, USA, 21 May 2011.
36. Decan, A.; Mens, T. What Do Package Dependencies Tell Us About Semantic Versioning? In *IEEE Transactions on Software Engineering*; IEEE: New York, NY, USA, 2019.
37. Raemaekers, S.; van Deursen, A.; Visser, J. Semantic versioning and impact of breaking changes in the Maven repository. *J. Syst. Softw.* **2017**, *129*, 140–158. https://doi.org/10.1016/j.jss.2016.04.008.
38. Conventional Commits. 2023. Available online: https://www.conventionalcommits.org/ (accessed on 23 May 2024).
39. Shahin, M.; Ali Babar, M.; Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In *IEEE Access*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2017; Volume 5, pp. 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629.
40. Decan, A.; Mens, T.; Mazrae, P.R.; Golzadeh, M. On the Use of GitHub Actions in Software Development Repositories. In Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 3–7 October 2022. https://doi.org/10.5281/zenodo.6634682.
41. Kinsman, T.; Wessel, M.; Gerosa, M.A.; Treude, C. How Do Software Developers Use GitHub Actions to Automate Their Workflows? In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021. Available online: http://arxiv.org/abs/2103.12224 (accessed on 5 May 2024).
42. Github Container Registry. 2023. Available online: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry (accessed on 23 May 2024).
43. Coquand, M. Evaluating Functional Programming for Software Quality in REST APIs. 2019 Available online: http://www.diva-portal.org/smash/get/diva2:1359684/FULLTEXT01.pdf (accessed on 14 January 2024).
44. Martin, R.C. *Agile Software Development, Principles, Patterns, and Practices*; Prentice Hall PTR: Hoboken, NJ, USA, 2014.
45. Moroz, B. Unit Test Automation of a React-Redux Application with Jest and Enzyme. 2019. Availablre online: https://www.theseus.fi/bitstream/handle/10024/184586/Moroz_Bogdan.pdf?sequence=2&isAllowed=y (accessed on 15 May 2024).

46. Raiküla, K.; Implementation of Automated End-To-End Testing in Web Applications. 2023. Available online: https://www.theseus.fi/bitstream/handle/10024/794423/Raikula_Karina.pdf?sequence=2 (accessed on 6 July 2024).
47. Jamil, M.A.; Arif, M.; Abubakar, N.S.A.; Ahmad, A. Software Testing Techniques: A Literature Review. In Proceedings of the 2016 6th international conference on information and communication technology for the Muslim world (ICT4M), 22–24 November 2016, pp. 177–182. https://doi.org/10.1109/ict4m.2016.045.
48. Reshma, S.G.; Mohan Kumar, H.P.; Manu, A.G. Smoke Test Execution in Software Application Testing. In Proceedings of the 4th International Conference on Emerging Research in Electronics, Computer Science and Technology, ICERECT, Mandya, India, 26–27 December 2022. https://doi.org/10.1109/ICERECT56837.2022.10059686.
49. Hawilo, H.; Jammal, M.; & Shami, A. Exploring Microservices as the Architecture of Choice for Network Function Virtualization Platforms. *IEEE Netw.* **2019**, *33*, 202–210. https://doi.org/10.1109/MNET.2019.1800023.
50. Wang, W. (2022). Research on Using Docker Container Technology to Realize Rapid Deployment Environment on Virtual Machine. In Proceedings of the 2022 8th Annual International Conference on Network and Information Systems for Computers, ICNISC, Hangzhou, China, 16–19 September 2022; pp. 541–544. https://doi.org/10.1109/ICNISC57059.2022.00112.
51. Oemig, F.; Blobel, B. A formal analysis of HL7 Version 2.x. *Stud. Health Technol. Inform.* **2011**, *169*, 704–708. https://doi.org/10.3233/978-1-60750-806-9-704.
52. P. Raghavan, H. Shachnai and M. Yaniv, Dynamic schemes for speculative execution of code. In Proceedings. Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.98TB100247), Montreal, QC, Canada, 1998, pp. 309-314, doi: 10.1109/MASCOT.1998.693711.
53. Nicholas, G. Cross-Origin Resource Sharing. Available online: http://edshare.soton.ac.uk/20595/ (accessed on 13 June 2024).
54. Nevedrov, D. Using JMeter to Performance Test Web Services. 2006. Available online: http://dev2dev.bea.com/lpt/a/509http://dev2dev.bea.com/pub/a/2006/08/jmeter-performance-testing.html (accessed on 13 June 2024).
55. Nguyen-Duc, A.; Do, M.V.; Luong Hong, Q.; Nguyen Khac, K.; Nguyen Quang, A. On the adoption of static analysis for software security assessment–A case study of an open-source e-government project. *Comput. Secur.* **2021**, *111*. https://doi.org/10.1016/j.cose.2021.102470.
56. Nkenyereye, L.; Jang, J.-W. Performance Evaluation of Server-side JavaScript for Healthcare Hub Server in Remote Healthcare Monitoring System. *Procedia Comput. Sci.* **2016**, *98*, 382–387. https://doi.org/10.1016/j.procs.2016.09.058.
57. Cordingly, R.; Yu, H.; Hoang, V.; Perez, D.; Foster, D.; Sadeghi, Z.; Hatchett, R.; Lloyd, W.J. Implications of Programming Language Selection for Serverless Data Processing Pipelines. In Proceedings of the 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, Canada, 17–22 August 2020.
58. Casalicchio, E.; Perciballi, V. Measuring Docker performance: What a mess!!! In Proceedings of the ICPE 2017-Companion of the 2017 ACM/SPEC International Conference on Performance Engineering, L'Aquila, Italy, 22–26 April 2017. https://doi.org/10.1145/3053600.3053605.
59. Bach-Nutman, M. Understanding The Top 10 OWASP Vulnerabilities. *arXiv* **2020**, arXiv:2012.09960.
60. Sharma, P. Securing Your Web Application A Deep Dive into OWASP Top 3 Security Risks. 2023. Available online: https://opencoursehub.cs.sfu.ca/bfraser/grav-cms/cmpt415/report/sample/OWASP_Top3SecurityRisks-HaitiHHA.pdf (accessed on 2 July 2024).
61. Ojamaa, A.; Düüna, K. Assessing the Security of Node.js Platform. In Proceedings of the 2012 International Conference for Internet Technology and Secured Transactions, London, UK, 10–12 Decembe 2012.