EUB
SuperHub

European Building Sustainability performance and energy certification Hub

D3.2 - Report on the test phase

This project has received funding from the European Union's H2020 research and innovation programme under Grant Agreement No. 101033916.

Project no. 101033916

Project acronym: EUB SuperHub

Project title: European Building Sustainability performance and energy certification Hub

Call: H2020-LC-SC3-B4E-4-2020

Start date of project: 01.06.2021.

Duration: 43 months

Deliverable title: D3.2 (Report on the test phase)

Due date of deliverable: 31. July 2024

Organisation name of lead contractor for this deliverable: FeliCITY-Tools Engineering Ltd.

Name	**Organization**
Bese Pál	FeliCITY-Tools Engineering Ltd.,

Dissemination level		
PU	Public	PU

Document history

History			
Version	Date	Reason	Revised by
01	14/07/2024	Draft for review	GEO
02	31/07/2024	Final version	FeliCity

Table of Contents

1 Introduction

1.1 Purpose of this document

The purpose of this document is to provide a comprehensive summary of the testing phase conducted prior to the system's launch for beta testing. This report aims to:

- Outline the scope and objectives of the testing activities.
- Detail the methodologies, tools, and environments used during the testing process.
- Present the test plan, including the schedule, test cases, and data preparation strategies.
- Describe the execution of various types of tests, such as unit, integration, system, and end-to-end testing.
- Summarise the results obtained from the testing phase, including identified defects and their severity.
- Highlight the issue management process, including defect tracking and resolution.
- Analyse the insights gained from the testing phase, focusing on root cause analysis, impact assessment, and lessons learned.
- Provide recommendations for immediate actions and future improvements to enhance the system's quality and reliability.
- Conclude with a summary of findings, final assessment, and the next steps to be taken.

This document serves as a reference for stakeholders to understand the testing efforts undertaken, the challenges encountered, and the overall readiness of the system for beta testing.

1.2 Scope of the testing

The scope of the testing phase for EUB SuperHub encompasses a thorough evaluation of the system's functionalities, performance, security, and user experience. The primary objectives of this testing phase are to identify and resolve any defects, ensure the system meets its requirements, and verify that it is ready for beta testing. Specifically, the scope includes:

- *Functional Testing*: Assessing each feature of the system to ensure it operates according to the requirements specified in D2.3. This includes verifying that all functions perform correctly and consistently under various conditions.
- *Performance Testing*: Evaluating the system's responsiveness, stability, and scalability. This involves stress testing, load testing, and performance benchmarking to ensure the system can handle expected and peak user loads without performance degradation.

- *Security Testing*: Identifying and addressing potential security vulnerabilities. This includes vulnerability scanning, and security code reviews to protect against unauthorised access, data breaches, and other security threats.
- *Usability Testing*: Ensuring the system provides a user-friendly experience. This includes evaluating the user interface, navigation, and overall user satisfaction to identify areas for improvement.
- *Compatibility Testing*: Verifying that the system functions correctly across different devices, browsers, and operating systems. This ensures a consistent user experience regardless of the platform used.
- *Regression Testing*: Ensuring that new code changes do not negatively impact existing functionalities. This involves re-running previously conducted tests to verify that the system remains stable and functional after updates.
- *Integration Testing*: Testing the interactions between different modules and components of the system to ensure they work together seamlessly. This is critical for identifying issues that may arise from module dependencies and interactions.
- *End-to-End Testing*: Conducting comprehensive tests that simulate real-world usage scenarios. This includes verifying the complete workflow from start to finish to ensure the system behaves as expected in a live environment.

The scope is designed to provide a holistic evaluation of the system, ensuring all key aspects are tested and validated before progressing to the beta testing phase. This approach aims to deliver a robust, reliable, and user-friendly system to end-users.

1.3 References

- Testing in .NET
- OpenAPI standard
- D2.3 - The EUB SuperHub Platform modules: Features and functions
- D2.4 - The digital logbook, definition of data requirements, sources, and collection process
- D3.1 – Report on the architecture and interoperability
- European Commission Digital Strategy
- GitLab
- NodeJS
- Redis in-memory database
- RabbitMQ messaging and streaming broker
- Jest – JavaScript Testing Framework
- MSTest - .NET Testing Framework
- BenchmarkDotNet
- Lighthouse testing
- Cypress – E2E Testing Framework

2 Definitions, acronyms, and abbreviations

CI/CD	Continuous integration and delivery
DBL	Digital building logbook
E2E	End-to-end testing
ECDS	European Commission Digital Strategy
HTTP	Hypertext Transfer Protocol
OIDC	OpenID Connect
PWA	Progressive web apps
REST	Representational state transfer
RESTful	API that conforms to the constraints of REST architectural style
SEO	Search engine optimisation
TDD	Test-Driven Development
WCAG	Web Content Accessibility Guidelines

3 Testing Objectives

The testing phase is an important component of the system development lifecycle, designed to verify that the system meets its specified requirements, defined primarily in the deliverables of D2.3 (*The EUB SuperHub Platform modules: Features and functions*) and D2.4 (*The digital logbook, definition of data requirements, sources, and collection process*), and is ready for deployment to beta testers. The objectives of this testing phase are to evaluate the system's functionality, performance, security, and usability. By establishing clear testing objectives, we aim to identify and address any defects or issues that could impact the system's effectiveness and reliability.

In this chapter, we will outline the specific goals and success criteria that guided our testing efforts.

This chapter will cover:

- The primary goals of the testing phase, detailing what we aim to achieve through our testing efforts.
- The success criteria that define the conditions under which the system is considered to have passed the testing phase.

3.1 Goals and Objectives

The primary goals and objectives of the testing phase are to confirm that the system is functional, reliable, and ready for deployment to beta testers. The testing phase aims to validate that the system meets its specified requirements and performs well under expected conditions. The specific goals and objectives are collected in Table 1.

No.	*Title*	*Description*
#1	Verify functionality	Ensure that all system features operate as intended and conform to the specified requirements.
#2	Measure performance	Assess the system's performance under normal and peak load conditions to ensure it meets the required performance benchmarks.
#3	Validate security	Conduct security testing to identify and mitigate potential vulnerabilities and threats. Check whether the system complies with relevant security standards and best practices.

#4	Enhance user experience	Evaluate the system's usability to confirm it provides an intuitive and user-friendly interface.
#5	Verify compatibility and integration	Test the system across various devices, browsers, and operating systems to ensure compatibility and consistent user experience. Verify that different system modules and components integrate seamlessly and function correctly together.
#6	Validate end-to-end workflows	Conduct end-to-end testing to simulate real-world usage scenarios and confirm that the system performs as expected in a live environment.
#7	Establish regression stability	Ensure that recent code changes do not negatively impact existing functionalities through rigorous regression testing. Confirm that the system remains stable and functional after updates and modifications.

Table 1. Goals and objectives of the testing phase

By achieving these goals and objectives, we aim to deliver a robust and high-quality system that meets user expectations and performs reliably in various conditions. This testing approach helps in identifying potential issues early, reducing risks, and ensuring a successful beta testing phase and subsequent full deployment.

3.2 Success Criteria

The success criteria for the testing phase are defined to guarantee that the system meets its quality standards and is ready for beta testing. These criteria provide a clear benchmark against which the outcomes of the testing activities can be measured. The success criteria include the following key aspects:

No.	*Title*	*Description*
#1	Functional completeness	All planned features and functionalities must be fully implemented and operational as defined in D2.3 and D2.4. Each feature should pass all its

		respective test cases without critical or major defects.
#2	Performance metrics	The system must meet predefined performance benchmarks, including response time, throughput, and resource utilisation.
#3	Security compliance	Security measures should comply with the following directives: • Web Application Security Standard C(2018) 7283 final • Web Applications Secure Development Guidelines Version 3 - EC DIGIT SECURITY ASSURANCE • IT Vulnerability and Remediation Management C(2018) 7284 final
#4	Usability standards	The system should provide an intuitive and user-friendly interface that meets user experience standards.
#5	Compatibility and integration	The system must be compatible with various devices, browsers, and operating systems as specified in the requirements. All system components and modules should integrate seamlessly, with no significant integration issues. Data should flow correctly between integrated components, maintaining consistency and accuracy.
#6	End-to-end functionality	End-to-end workflows should be tested and validated to ensure they operate smoothly from start to finish.
#7	Regression stability	Regression tests should confirm that recent changes or updates do not introduce new defects or negatively impact existing functionalities.

Table 2. Success criteria

4 Testing Methodology

This chapter covers the overall strategy, test environment setup, tools and software utilised, and roles and responsibilities within the testing team.

Understanding the testing methodology provides insight into the planning, execution, and management of testing activities. By detailing the testing methodology, we aim to demonstrate the rigor and thoroughness of our testing process, ensuring the system's readiness for beta testing.

4.1 Overview of the System

EUB SuperHub, as a software, is built on a microservices architecture, which allows for modular development and scalability. Each microservice is designed to handle specific functions, working together to provide the overall functionality of the system. The key microservices playing a critical role in the system are summarised below.

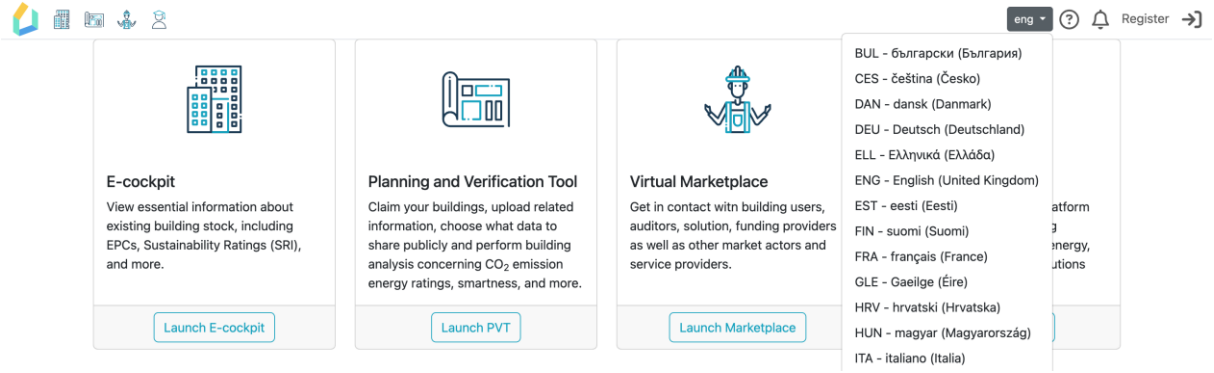

Figure 1 Home screen of the EUB SuperHub

4.1.1 FeliCity API

The FeliCity API is responsible for building-related geometry and energy calculations. This service performs complex computations required for managing building data, such as calculating energy consumption, analysing building geometries, and providing data (e.g., climate data, building usage profiles) for further processing and visualisation.

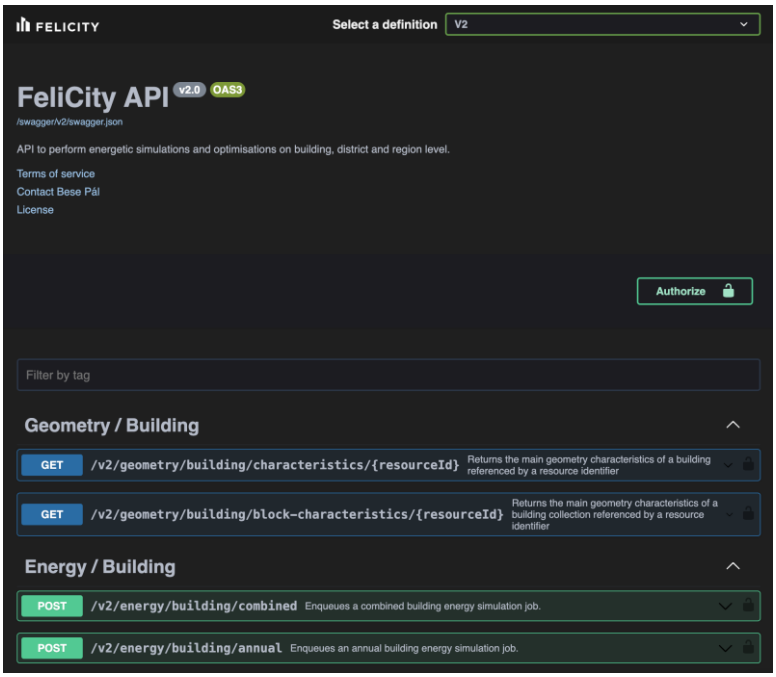

Figure 2 OpenAPI documentation of FeliCIty API

4.1.2 FeliCity Identity Provider

The FeliCity Identity Provider is an OpenID Connect (OIDC) server used for authenticating users. It manages user identities and guarantees secure access to the system by providing authentication tokens, enforcing security policies, and integrating with other services to facilitate user management.

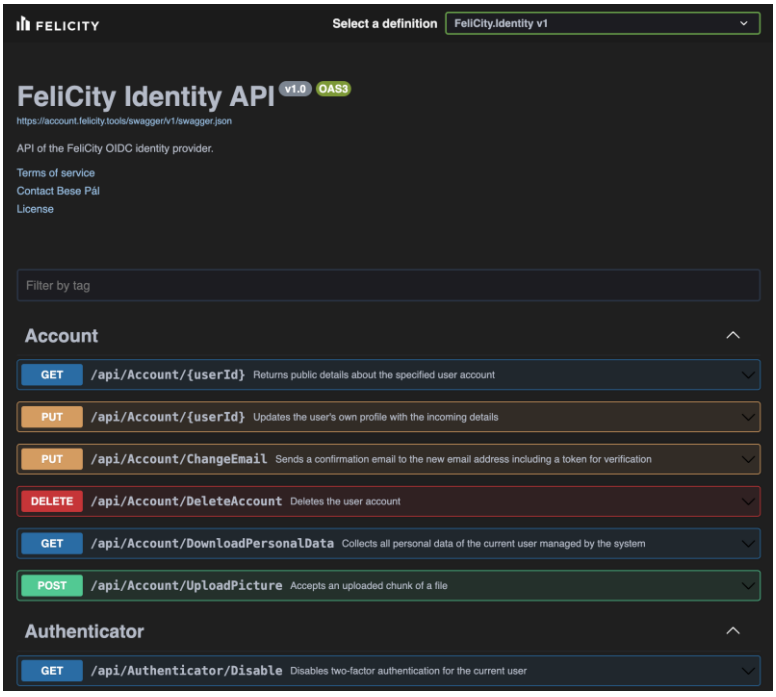

Figure 3 OpenAPI documentation of the FeliCity Identity API

4.1.3 Digital Building Logbook

The Digital Building Logbook (DBL) service stores and manages comprehensive data about buildings. This includes maintenance records, historical data, compliance documentation, and other relevant information that contributes to the lifecycle management of building assets as defined in the deliverable D2.4.

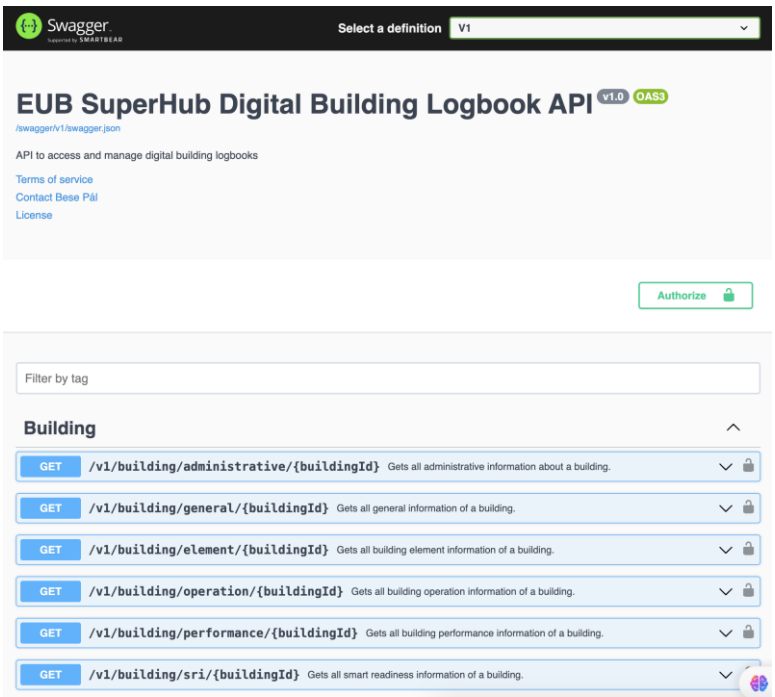

Figure 4 OpenAPI documentation of the EUB SuperHub DBL API

4.1.4 EUB SuperHub Web Service

The EUB SuperHub application itself is also a web service and acts as an interface for integrating external data sources and services. It enables the system to communicate with other platforms, exchange data, and provide additional functionalities through APIs, enhancing the system's interoperability and data richness.

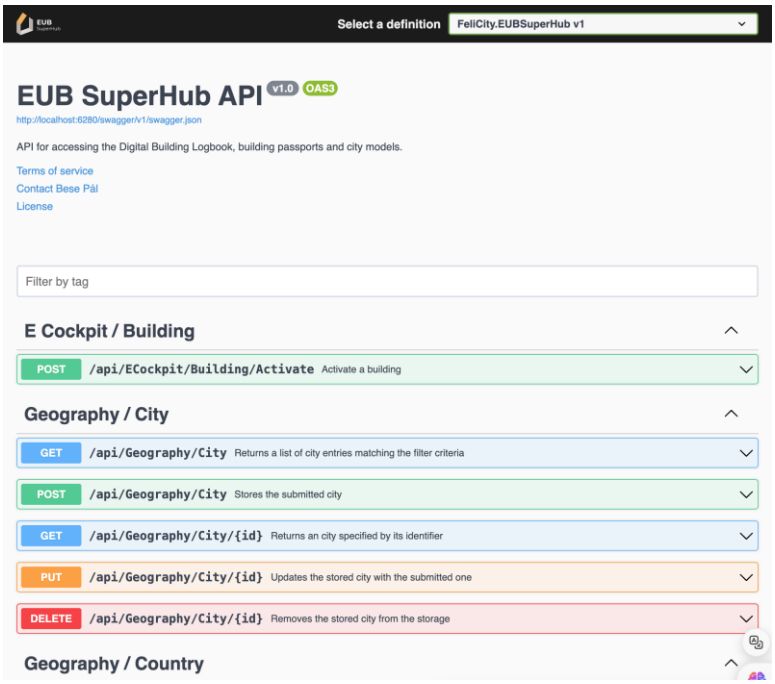

Figure 5 OpenAPI documentation of the EUB SuperHub API

4.1.5 FeliCity Cache Server

The FeliCity cache server is a Redis-based central service used for temporal data storage. It improves the system's performance by caching frequently accessed data, reducing database load, and providing faster access to transient data required for calculations and real-time processing.

This service is essential for complying with the constraints of a RESTful API. For instance, when a building energy calculation is requested, the building configuration is submitted as a HTTP POST request to create a resource on the server. This resource is assigned to a universally unique identifier that is generated by the cache server. This resource is then picked up by a calculation unit, when the job manager distributed the calculation task.

Database Alias	Host:Port	Connection Type	Modules	Last connection ↑
FeliCity cache server	cache.felicity.local:...	Standalone		1 minute ago

Figure 6 FeliCity cache server in Redis Insight

4.1.6 RabbitMQ-Based Queuing Service

The RabbitMQ-based service handles job distribution within the system. It facilitates asynchronous communication between microservices, ensuring that tasks such as data processing, notifications, and other background jobs are efficiently managed and executed without impacting the system's responsiveness.

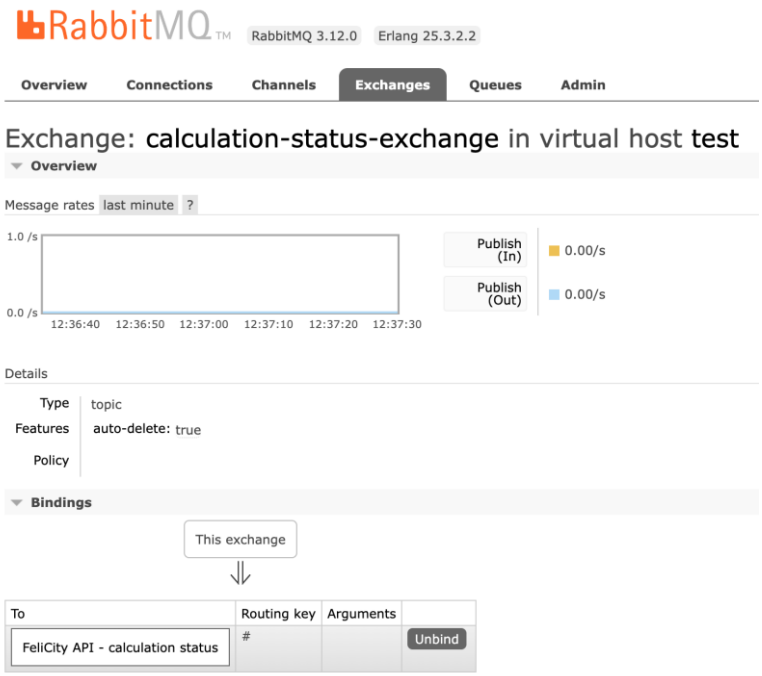

Figure 7 Exchange used for broadcasting status of simulations

4.1.7 Calculation units

Calculation intensive tasks, like building energy simulations, are performed asynchronously when the capacity is available. Arbitrary number of calculation units can join to the job queue for performing calculations in a parallel manner.

4.2 Testing Strategy

The testing strategy for the system is designed to reach comprehensive coverage and validation of all microservices and their interactions. The strategy encompasses various types of testing, each targeting different aspects of the system to ensure it meets the required standards of functionality, performance, security, and usability.

The specific testing strategies need to be in line with the system's architecture and follow how it separates duties into classes, packages, services, and applications.

4.2.1 Unit Testing

Unit testing focuses on validating the functionality of individual components or modules within each microservice. By isolating each unit, we can verify that it performs correctly in various scenarios. This helps in identifying and fixing bugs early in the development process. Each service, such as the FeliCity API, Identity Provider, DBL, and others, undergoes rigorous unit testing to verify their individual functionalities.

4.2.2 Integration Testing

Integration testing examines the interactions between different services to ensure they work together seamlessly. Given the microservice architecture, it's crucial to validate the communication pathways, data flow, and dependency management between services like the FeliCity API, cache server, RabbitMQ-based service, the DBL, and others.

4.2.3 System Testing

System testing involves testing the entire system to confirm it meets the specified requirements. This includes testing end-to-end workflows and verifying that all integrated components function together as expected. System testing helps in identifying issues that may not be apparent in unit or integration testing, such as workflow interruptions or data inconsistencies.

4.2.4 Performance Testing

Performance testing assesses the system's behaviour under various load conditions. This includes stress testing, load testing, and scalability testing. Key areas of focus are the system's response time, throughput, and resource utilisation, particularly for services like the FeliCity API and cache server, which handle significant data processing and storage operations.

4.2.5 Security Testing

At present, FeliCity does not have automated security testing integrated into its CI/CD pipelines. Instead, security testing efforts are focused on manual verification processes to ensure the system's alignment with the EC Digital Strategy (ECDS). This involves ensuring that the system adheres to the following key guidelines:

- *Web Application Security Standard C(2018) 7283 final*: Ensuring web applications meet security standards to prevent common vulnerabilities such as SQL injection, cross-site scripting (XSS), and other exploits.
- *Web Applications Secure Development Guidelines Version 3 - EC DIGIT SECURITY ASSURANCE*: Following secure development practices to build robust and secure applications.
- *IT Vulnerability and Remediation Management C(2018) 7284 final*: Identifying, managing, and mitigating vulnerabilities to minimize the risk of security incidents.

4.2.6 Usability Testing

Usability testing evaluates the system's user interface and overall user experience. This involves gathering feedback from real users to identify any

usability issues and areas for improvement. Ensuring a user-friendly interface is critical for user acceptance and satisfaction.

4.2.7 Compatibility Testing

Compatibility testing ensures that the system works correctly across different devices, browsers, and operating systems. This is important for providing a consistent user experience regardless of the platform used. The EUB SuperHub web service and other web-based components are tested extensively for compatibility.

4.2.8 Regression Testing

Regression testing guarantees that recent changes or updates do not negatively impact existing functionalities. This involves re-running previously conducted tests to verify that the system remains stable and functional after modifications. It is crucial for maintaining the integrity of the system during continuous development and deployment.

4.2.9 Test-Driven Development

Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. During the development of the EUB SuperHub, FeliCity adopted TDD to enhance the robustness and maintainability of the system.

The TDD process followed at FeliCity included:

- Writing unit tests for each new feature or functionality before writing the corresponding code.
- Running all tests frequently to ensure new changes do not introduce any regressions.
- Refactoring the code based on test outcomes to improve design and maintainability.

4.3 Test Environment

The test environment provides a controlled setting where the system can be evaluated under conditions that closely mimic the production environment. For the system built on microservices, the test environment is designed to ensure each microservice is tested individually and in conjunction with others to validate their interactions and overall system performance.

4.3.1 Infrastructure Setup

The test environment is hosted on a robust and scalable infrastructure that mirrors the production setup. This includes multiple virtual machines configured to replicate the distributed nature of the microservices architecture. Each microservice is deployed on this infrastructure to provide realistic testing conditions.

4.3.2 Configuration Management

Configuration management is handled across multiple levels to ensure consistency, security, and efficiency. Different configuration management techniques are employed for each test type to cater to the specific requirements of unit, integration, and system testing.

For unit and integration testing, each package includes a sample configuration file. These configurations are tailored specifically for development and testing environments, ensuring that automated testing infrastructure can execute tests defined within the package in isolated contexts.

In staging and production environments, configuration management is streamlined and secured through GitLab. During the automated deployment process, GitLab handles the configuration values. This guarantees that sensitive configuration details are managed securely, reducing the risk of exposure and ensuring that only the correct, environment-specific settings are applied.

4.3.3 Test Data

Each package within the system defines its own data model and validation logic, making it essential for each package to independently handle the generation of test data. This approach confirms that the test data aligns with the specific requirements and constraints of each package, maintaining consistency and reliability across the testing process.

To achieve this, each package is responsible for generating both valid and purposely invalid test data to thoroughly test its functionality and validation rules. The chosen technique to facilitate this is the implementation of a required *TestDataFactory* service within each package. The *TestDataFactory* service standardizes the creation of test data, ensuring that it meets the package's specific data model and validation criteria.

4.3.4 Monitoring and Logging

In line with standard .NET practices, logging is implemented using the ILogger interface, providing flexible access to various logging providers. This approach ensures that logs can be consistently and effectively managed across different environments and scenarios.

Logging is performed at multiple levels to capture a wide range of information:

- *Debug*: Detailed information useful during development and testing.
- *Information*: General operational events to track the system's workflow.

- *Warning*: Indications of potential issues that do not immediately affect system functionality.
- *Error*: Events that indicate significant problems that need to be addressed.
- *Critical*: Severe issues that require immediate attention and could lead to system failure.

During testing, the logging level is set to *Debug* to capture detailed information about the system's operation, facilitating thorough analysis and troubleshooting. In the production environment, the logging level is set to *Error* to focus on significant issues, ensuring that critical problems are promptly identified without overwhelming the logs with less pertinent information.

4.3.5 Network Configuration

Network configuration in the test environment is tailored to different scenarios to reach optimal testing conditions. For automated integration tests of individual packages, a basic network setup is used in an isolated local environment. This simplifies testing, focusing on package interactions without external network interference.

When testing APIs and applications, a local network configuration is used without firewalls or load balancers. This provides a direct communication path, allowing for accurate assessment of core functionality and performance. Starting with these simplified network settings ensures fundamental issues are resolved early, facilitating more comprehensive testing in realistic network conditions later.

The network setup in the staging environment replicates the production network configuration, including firewalls, load balancers, and network segmentation. This way network-related issues, such as latency, bandwidth constraints, and security vulnerabilities, can be identified and addressed during testing.

4.3.6 Security Measures

Testing environments can inadvertently expose security flaws if they are not protected. These vulnerabilities can be exploited to gain access to the development or production environments, posing significant risks to the overall system. Security measures applied to the test environment are as follows:

4.3.6.1 Restricted Access

The test environment is only accessible from within the FeliCity corporate network. This way external access is blocked, reducing the risk of unauthorised access and potential security breaches. By limiting access to

trusted internal users, we can maintain tighter control over the testing environment.

4.3.6.2 Two-Factor Authentication

Two-factor authentication (2FA) is enforced for all users accessing the test environment. This additional layer of security requires users to provide two forms of verification before gaining access, significantly enhancing security by making it more difficult for unauthorised users to access the environment, even if they have obtained a valid password.

4.3.6.3 Ephemeral Test Data and Data Clean-Up

Generated test data is not stored after tests are completed. Each test case includes a clean-up method that removes all test data used by the test upon completion. This combined approach ensures that any potentially sensitive or confidential data used during testing does not persist beyond the testing session, thereby reducing the risk of data leakage or unauthorized access to test data. Maintaining a clean state for subsequent tests prevents any potential data conflicts or security issues arising from leftover test data.

4.3.6.4 Secure Test Logs

Test logs are stored on a virtual machine that is not accessible from the internet, so sensitive log information remains secure and is only accessible to authorised personnel within the FeliCity network. By preventing internet access, we mitigate the risk of log data being exposed or compromised.

4.3.7 Backup and Recovery

Ensuring backup and recovery mechanisms is for maintaining the integrity and availability of the test environment. Test cases are designed not only for the initial testing period preceding beta testing but also for future validations. This forward-thinking approach necessitates reliable backup and recovery processes to handle potential data loss and system failures.

Test and reference data that are not generated on-the-fly during tests are backed up to prevent loss in case of test failures. This precaution allows for quick restoration of data, ensuring continuity in testing and preventing disruptions that could arise from data corruption or loss. Having backup mechanisms in place also supports safe rollbacks to previous versions when necessary, providing a safeguard against unexpected issues during testing.

Each virtual machine in the test environment is backed up separately, regularly, and incrementally. Databases are backed up daily to make sure that all data changes are preserved. Daily backups provide a reliable recovery point, allowing for minimal data loss if restoration is required.

In addition to virtual machines and databases, the source code is backed up along with the entire GitLab system, making all aspects of the development and testing environment recoverable.

4.4 Test Tools and Software Used

The testing phase leverages a variety of tools and software to ensure comprehensive coverage and efficient tracking of testing activities. The selection of these tools is tailored to the specific needs of the system's microservices architecture, ensuring that both server-side and client-side components are thoroughly tested.

By leveraging these tools and frameworks, we set up a robust and comprehensive testing process. Each tool is selected based on its strengths and suitability for the specific testing requirements of the system's microservices architecture. This approach helps in identifying and addressing issues early in the development cycle, ensuring that the system is reliable, performant, and ready for deployment.

4.4.1 GitLab for Development and Tracking

The entire development process is organised and tracked using FeliCity's own GitLab instance. GitLab provides a robust platform for version control, issue tracking, and continuous integration/continuous deployment (CI/CD). This way all code changes are systematically tracked, and any issues identified during testing are documented and managed efficiently. The use of GitLab facilitates collaboration among team members, enabling integration of development and testing activities.

4.4.2 MSTest Framework for Server-Side Testing

For server-side unit and integration testing in the .NET environment, Microsoft's MSTest framework is utilised. MSTest provides a comprehensive set of tools for writing and running tests, enabling developers to validate the functionality of individual components and their interactions within the system. Key features of MSTest include:

- Test case creation and execution
- Assertions to validate expected outcomes
- Test results reporting and analysis, including coverage reporting

4.4.3 Jest for Client-Side Testing

For client-side unit and integration testing in the NodeJS environment, Jest is the testing framework of choice. Jest offers a powerful and easy-to-use platform for testing JavaScript applications, providing features such as:

- Snapshot testing to ensure UI consistency
- Mocking and spies to simulate various scenarios
- Built-in code coverage reporting

Jest is employed to validate client-side logic and interactions, ensuring that user interfaces and other client-side components function correctly and deliver a seamless user experience.

4.4.4 BenchmarkDotNet for Performance Testing

To evaluate the performance of the API, the BenchmarkDotNet library is used. BenchmarkDotNet is a powerful and flexible tool for benchmarking .NET applications, offering features such as:

- Precise and accurate performance measurements
- Detailed reporting and analysis of benchmark results
- Support for various performance metrics, including execution time, memory usage, and throughput

Using BenchmarkDotNet, we conduct performance tests on the FeliCity API to verify it meets the required performance benchmarks and can handle expected loads efficiently.

4.4.5 Lighthouse testing

Lighthouse testing is integrated into our CI/CD pipeline to evaluate the performance, accessibility, best practices, SEO, and PWA capabilities of our web applications. Using Google's open-source tool, we automatically generate detailed reports to identify areas for improvement.

Lighthouse measures key performance indicators such as load times and interactivity, checks accessibility against WCAG standards, and evaluates adherence to best practices and SEO. For progressive web apps, it assesses service worker implementation and offline functionality.

By incorporating Lighthouse testing, we continuously monitor and enhance the performance, accessibility, and overall quality of our web applications, ensuring they meet high standards and deliver excellent user experiences.

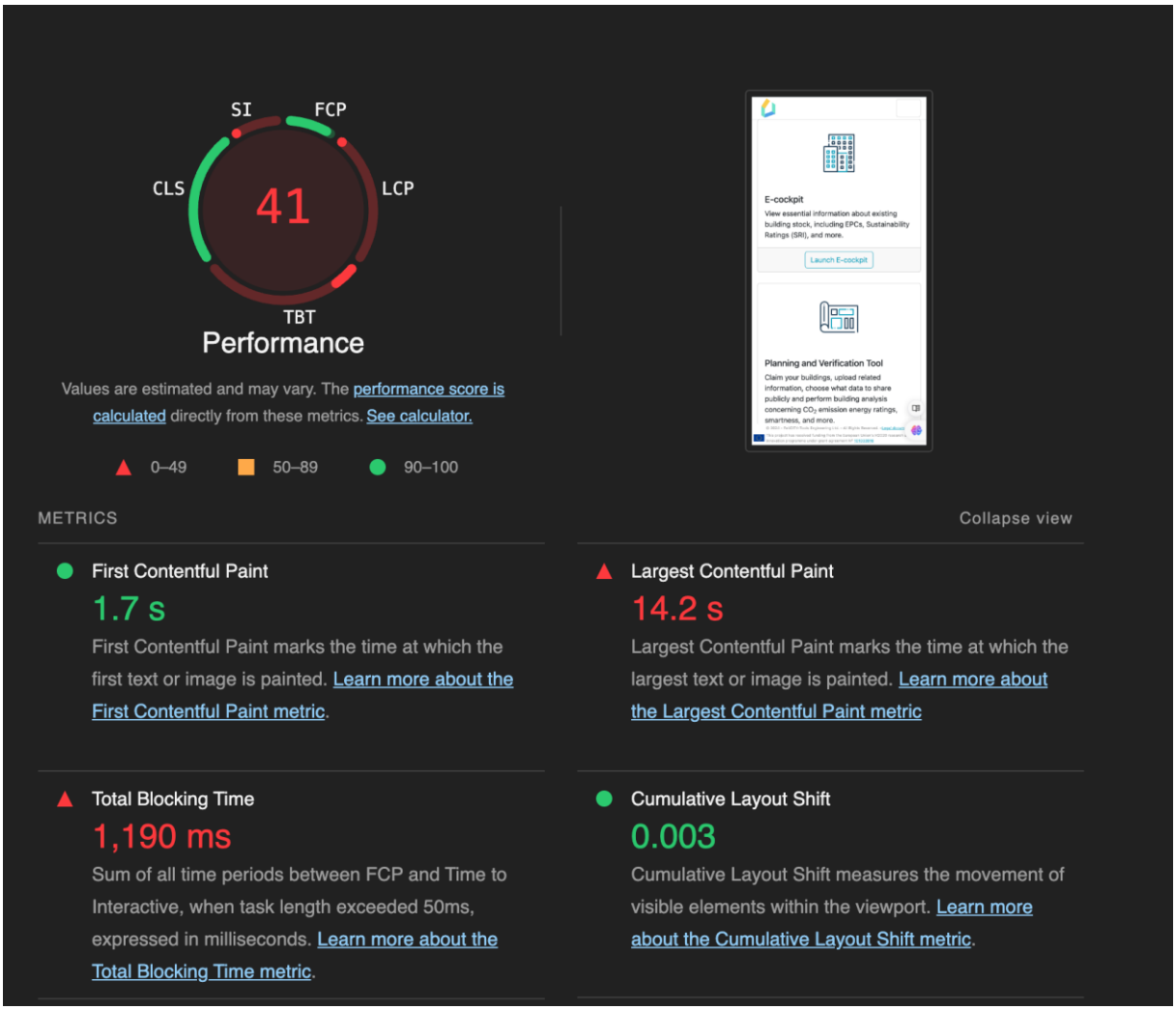

Figure 8 Lighthouse report showing low score for mobile tests

4.4.6 Cypress test cases

For end-to-end (E2E) testing, Cypress is integrated into our CI/CD pipeline, automatically running tests with every code commit and merge request. Tests are organized in a clear structure, written in JavaScript, and placed in a specific directory in the code repository. They simulate user actions with commands like *cy.visit()*, *cy.get()*, and *cy.click()*, and verify outcomes using assertions.

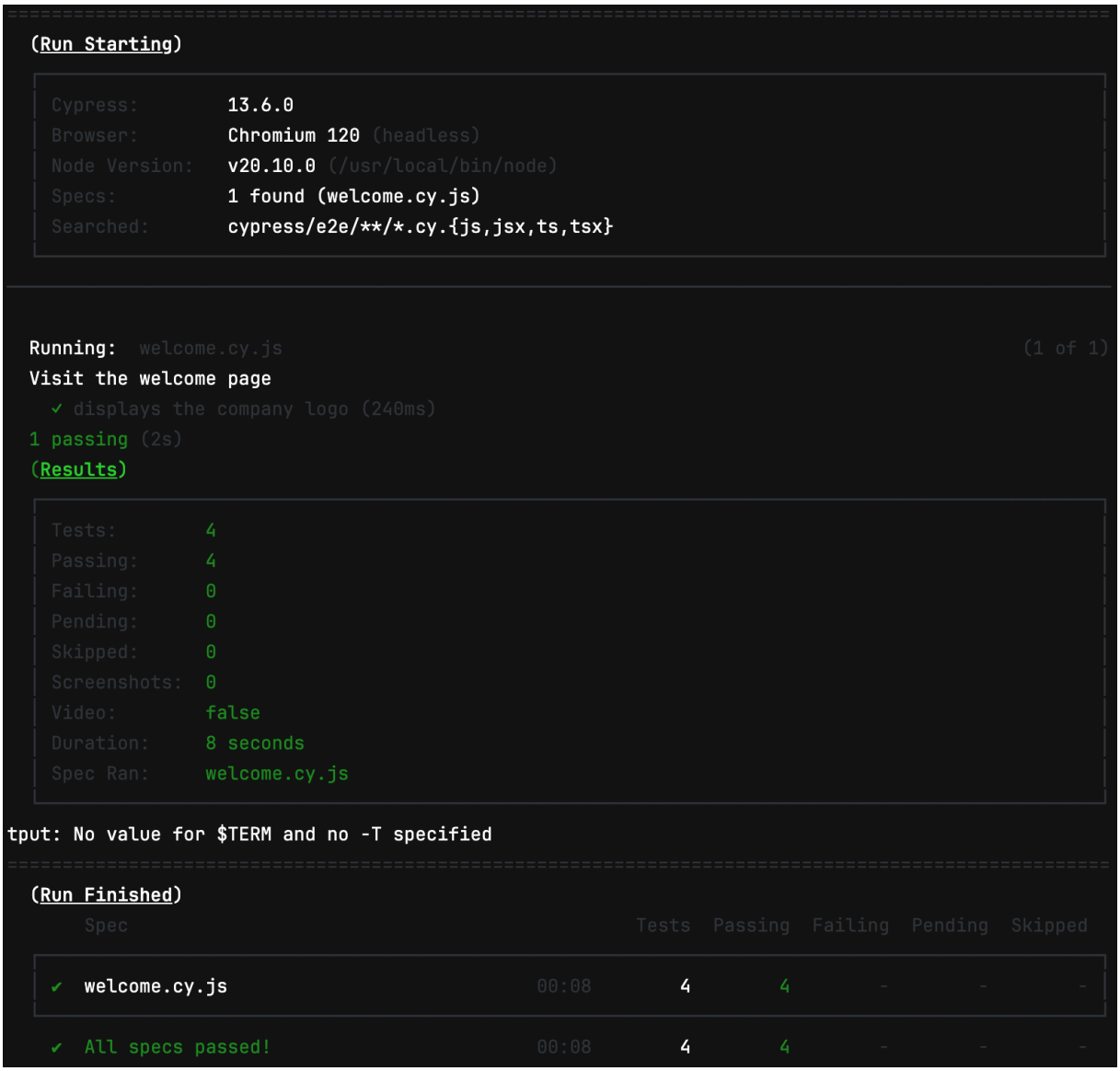

Figure 9 E2E testing using Cypress

4.5 Roles and Responsibilities

FeliCity's small team faces significant challenges in testing due to the lack of dedicated resources. Ideally, distinct roles for testing would ensure thorough coverage, but our limited size means that team members must juggle both development and testing tasks. This is a notable shortcoming that impacts our effectiveness.

Developers handle unit and integration tests for their modules, which can dilute their focus and compromise testing depth. System and end-to-end testing are shared among the team, leading to potential gaps in coverage and missed defects. This lack of dedicated testing resources remains a significant constraint, affecting the thoroughness and reliability of our testing process.

5 Test Plan

In this chapter, we outline the test plan designed to guide the testing phase of the system. The test plan serves as a blueprint, detailing the schedule, development of test cases, and preparation of test data. It guarantee that all testing activities are systematically organised and executed, providing a clear framework for evaluating the system's functionality, performance, security, and usability.

5.1 Test Schedule

The test schedule was structured to support validation of each component and the overall system within a defined timeline. Unit and integration testing for both client-side and server-side components were conducted concurrently with the implementation of specific components/packages. This continuous testing approach allowed for early detection and resolution of defects, ensuring that each component was validated before moving on to the next phase.

In the final phase, spanning three weeks, the focus shifted to performance and end-to-end testing. During this period, performance tests were designed and specified. Simultaneously, end-to-end tests were developed to simulate real-world usage scenarios.

5.2 Test Case Development

For both client-side and server-side components, test cases were designed to cover a wide range of scenarios, including typical usage, edge cases, and error conditions. Each test case included detailed steps, expected outcomes, and criteria for success, facilitating systematic validation of each component; following the *Arrange-Act-Assert* approach.

5.3 Test Data Preparation

The preparation of test data is a central aspect of the testing process, ensuring that the system is evaluated under realistic and varied conditions. FeliCity chose to use randomly generated data for each test case and run, which offers significant advantages in terms of testing coverage and robustness. By utilising random data, we can simulate a wider range of scenarios, identify potential edge cases, and ensure that the system performs reliably under diverse conditions.

For the test data to be effective, it must be semantically valid unless the test case specifically aims to check the system's behaviour with invalid data. This means that randomly generated data must adhere to the logical rules and constraints of the domain. For instance, when testing the DBL, renovation dates must always be after the building's construction date. Ensuring

semantic validity in the test data helps in accurately assessing the system's functionality and prevents false positives that could arise from using illogical data.

Maintaining valid relationships among collections of data is another critical requirement. Random data generation must account for interdependencies within the data set. For example, ensuring that energy consumption data aligns with the building's specifications and usage patterns. By preserving these relationships, the test data closely mirrors real-world scenarios, providing more reliable and meaningful test results.

Despite the advantages of random data generation, some data cannot be generated randomly due to the need for consistency and reference accuracy. For instance, reference data such as country names, currencies, and units of measure must remain consistent and accurate across all test cases. In these instances, FeliCity's own test databases were utilised to provide the necessary reference data. This way reference data is always correct and consistent, avoiding discrepancies that could skew the test results.

Using a combination of randomly generated data and fixed reference data allows us to achieve comprehensive testing coverage without needing an impractically large dataset. Randomly generated data introduces variability and helps uncover edge cases, while fixed reference data ensures consistency where it is crucial. This balanced approach allows for thorough testing of the system's functionality, performance, and reliability across a wide range of conditions.

Overall, the strategy of using semantically valid, randomly generated data, combined with fixed reference data, provides a robust framework for testing. It enables us to simulate realistic and varied scenarios, ensuring that the system is rigorously evaluated and ready for deployment.

6 Test Execution

The test execution phase is designed to verify that all tests are run continuously and efficiently, leveraging GitLab pipelines. Each merge request and version tag triggers the execution of all relevant tests, ensuring that any changes to the codebase do not introduce new defects.

Unit and integration tests are executed separately for each package, providing focused validation of individual components. At the application level, functional testing of endpoints, performance testing, and Lighthouse testing for web performance and accessibility are performed automatically. This testing strategy allows all aspects of the system are thoroughly

evaluated, from individual units to the overall application performance and user experience.

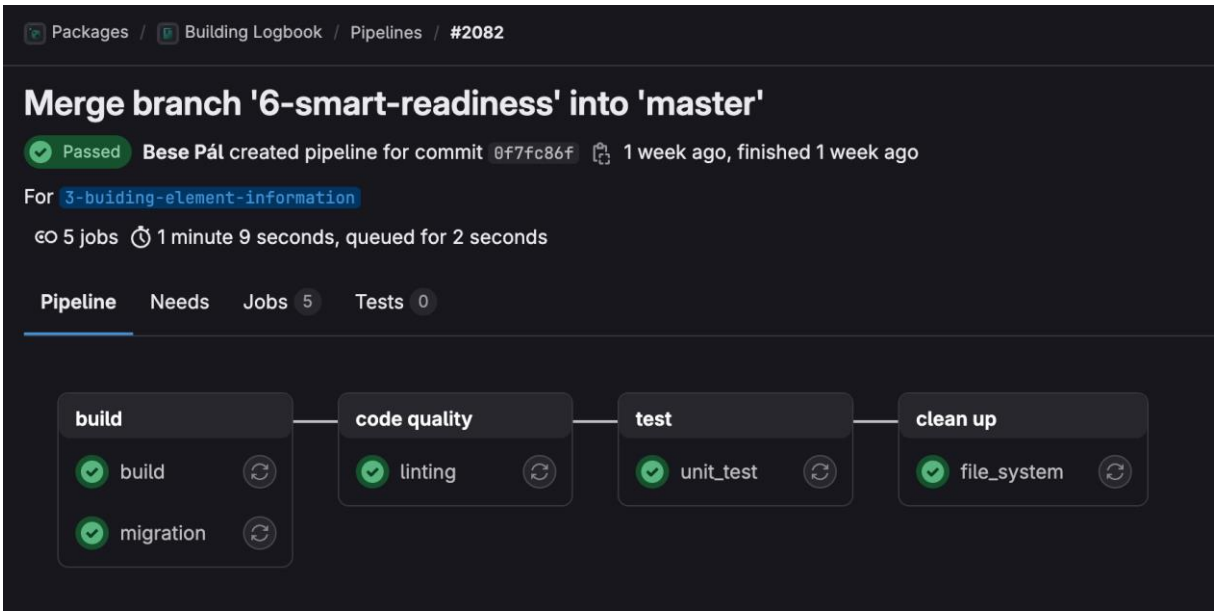

Figure 10 Pipeline that run on the SRI merge request of the DBL

6.1 Test Execution Process

The test execution process is a structured and automated workflow designed to ensure thorough and continuous testing of the system. By integrating testing into the GitLab CI/CD pipelines, FeliCity can make sure that each code change is rigorously validated before being merged into the main codebase or deployed to production.

6.1.1 Continuous Integration and Testing

Every merge request and version tag in the GitLab repository triggers a pipeline that includes various stages of testing. This pipeline is configured to run a comprehensive suite of tests to catch defects early and maintain the quality of the codebase. The continuous integration process helps in identifying issues as soon as they are introduced, reducing the time and effort required for debugging and fixing them later in the development cycle.

6.1.2 Unit and Integration Testing

Unit tests are executed for each individual package, focusing on validating the smallest components of the system, verifying that each unit performs as expected in isolation. Integration tests, on the other hand, are conducted to verify the interactions between different units within each package. This approach ensures that components not only function correctly on their own but also work seamlessly together.

6.1.3 Application-Level Testing

At the application level, a series of automated tests are performed to validate the system's overall functionality and performance. Functional testing of the API endpoints checks if the system's interfaces work correctly and return the expected results. Performance testing, conducted using the BenchmarkDotNet library, measures the system's responsiveness and stability under various load conditions. Additionally, Lighthouse testing is used to evaluate web performance, accessibility, and best practices, ensuring that the application provides a high-quality user experience.

6.1.4 Execution and Reporting

The results of each test run are automatically reported back to the GitLab interface, providing immediate feedback to developers. Any failed tests or detected defects are logged, and developers are notified to address the issues promptly. This automated feedback loop allows development team to maintain a high level of code quality and quickly respond to any problems that arise.

By incorporating continuous testing into the development workflow, FeliCity ensures that its system is robust, reliable, and ready for deployment. The automated test execution process not only saves time and resources but also enhances the overall quality and stability of the system, providing confidence in its readiness for beta testing and production use.

6.2 Types of Tests Conducted

6.2.1 Unit Testing

Unit testing forms the foundation of the testing process, focusing on the smallest, individual components of the system to confirm they function correctly in isolation. Each unit, typically a function or method within a component, is tested independently to verify its behaviour and output against expected results. This granular approach allows developers to identify and fix issues at the earliest stage of the development process, ensuring that each piece of code performs as intended before it is integrated with other components.

For the server-side components developed in the .NET environment, unit tests are written and executed using Microsoft's MSTest framework. MSTest provides a suite of tools for defining, running, and analysing tests, making it easier to validate the functionality of individual methods and classes. Each unit test includes specific inputs and expected outputs, along with assertions to confirm the correctness of the code's behaviour.

On the client-side, particularly within the NodeJS environment, unit tests are written using Jest. Jest is a powerful testing framework designed to work

seamlessly with JavaScript, offering features such as mocking, spies, and snapshot testing. By leveraging Jest, developers can create detailed unit tests that cover various scenarios and edge cases, ensuring the robustness of client-side logic and interactions.

Unit testing is conducted continuously throughout the development process. Each time a new feature is added, or an existing one is modified, the corresponding unit tests are updated and re-executed. This continuous validation helps maintain a high level of code quality and prevents regression issues, ensuring that new changes do not inadvertently break existing functionality.

6.2.2 Integration Testing

Integration testing is a crucial step in the testing process, designed to ensure that different units and components of the system work together seamlessly. While unit testing verifies individual units in isolation, integration testing focuses on the interactions between these units, identifying any issues that may arise when they are combined.

For the server-side components, integration tests are conducted using the MSTest framework. These tests validate the interactions between various modules and services, such as the FeliCity API, Identity Provider, and cache server. Integration testing verifies that data flows correctly between these components, that API endpoints are correctly implemented, and that the overall system behaves as expected when different parts are integrated. For instance, when testing the DBL, integration tests confirm that data from the FeliCity API is correctly processed and stored, and that user authentication via the Identity Provider functions properly within the workflow.

On the client-side, integration tests are executed using Jest. These tests verify that different parts of the user interface and client-side logic interact correctly. This includes ensuring that data fetched from server-side APIs is accurately processed and manipulated within the client application, and that user actions trigger the appropriate client-side processes and updates. For example, one integration test checks that the submission of building element information correctly triggers the DBL API call, and processes the response.

Integration testing is performed continuously as part of the CI/CD pipeline in GitLab. Each merge request triggers a suite of integration tests to validate that recent changes do not disrupt existing interactions between components.

6.2.3 System Testing

System testing evaluates the integrated system to confirm it meets all specified requirements for functionality, performance, and usability.

Functional testing verifies the operation of user interfaces, APIs, and component interactions, while performance testing using BenchmarkDotNet measures system responsiveness and scalability under various loads.

6.2.4 End-to-end testing

End-to-end testing validates the complete workflow of the system, ensuring that all integrated components function seamlessly together in real-world scenarios. This testing involves simulating user activities from start to finish, such as logging in via the FeliCity Identity Provider, submitting building data through the FeliCity API, processing this data in the Digital Building Logbook, and displaying the results.

6.3 Test Coverage

Test coverage measures the proportion of code executed during tests, helping assess test thoroughness. There are various methods for measuring coverage, including line, branch, and method coverage.

Line coverage, which measures the percentage of executed lines of code, is the easiest to achieve but doesn't guarantee comprehensive testing. Branch coverage, measuring the percentage of decision points tested, offers better insight by ensuring all possible outcomes are covered.

FeliCity has set a 70% threshold for branch coverage to better represent real-life scenarios and edge cases, while maintaining over 90% of line coverage, leading to more comprehensive tests, improving code resilience and overall system quality.

Module	Line	Branch	Method
FeliCity.Geography	98.95%	71.05%	97.24%

	Line	Branch	Method
Total	98.95%	71.05%	97.24%
Average	98.95%	71.05%	97.24%

Table 3. Coverage report for FeliCity.Geography package

7 Test Results

The testing phase of the EUB SuperHub project has yielded comprehensive results that validate the system's functionality, performance, security, and usability. Overall, the system's services performed well across various testing categories, demonstrating its readiness for beta testing. Functional tests confirmed that all features operate as intended, with minor discrepancies noted in specific edge cases. Performance tests, revealed that the system meets the required benchmarks for response time and resource utilisation.

7.1 Defects Identified

During the testing phase, several defects were identified and categorised based on their impact and urgency. A total of 37 defects were identified, with the following distribution:

- Critical: 0
- Major: 12
- Minor: 25

No critical defects were identified, indicating that the system is free from issues that could cause system crashes, thanks to the test-driven development approach applied. Major defects included performance bottlenecks under peak load conditions. Minor defects were mainly related to coding standards, API documentation, and parameter binding.

7.2 Severity and Priority of Defects

Each defect was assessed for its severity and priority, ensuring that high-impact issues were addressed first. The severity and priority were defined as follows:

- Critical (High Priority): Defects causing system crashes, data corruption, or significant security vulnerabilities.
- Major (Medium Priority): Defects affecting system performance, usability, or integration, but not causing immediate operational failure.
- Minor (Low Priority): Defects with minimal impact on functionality, often cosmetic or related to non-critical features.

8 Issue Management

Effective issue management is critical to maintaining the quality and reliability of the system. Our approach integrates tightly with the designated GitLab CI/CD pipelines, ensuring that no source code is delivered if any related tests fail. This rigorous process ensures that code changes do not introduce new defects and that the system remains stable and functional.

8.1 Defect Tracking Process

The defect tracking process is integrated into the GitLab CI/CD pipelines. Whenever a test fails, whether during unit, integration, system, or end-to-end testing, a GitLab report is created. This includes details about the failed test, the nature of the defect, and any relevant logs or screenshots.

8.2 Resolution and Retesting

Once an issue is created, a corresponding draft merge request is generated. This draft includes the necessary code changes aimed at resolving the defect. However, this merge request can only be merged if it meets all code quality standards and passes the required unit and integration tests. This process ensures that fixes are thoroughly vetted and validated before being integrated into the main codebase. In some instances, the unit test itself may need to be revised to correctly reflect the desired functionality and provide accurate testing.

8.3 Alert integration

Despite achieving 100% branch coverage in tests, errors may still occur in the production environment. To address this, we have implemented GitLab alert integration, which ensures that all exceptions are logged and reported automatically. This integration helps in promptly identifying and responding to issues that arise in the production environment, maintaining high system reliability and user satisfaction.

Severity	Start time ↓	Alert	Events	Incident	Assignees	Status
◆ High	2 days ago	#1 Country not found	4	None		Triggered

Table 4. Alert that shows an exception that occurred during the energy simulation

9 Analysis and Insights

9.1 Root Cause Analysis

During the recent testing phase, several defects were identified, primarily related to integration issues and data validation errors. For example, a significant number of defects were traced back to inconsistencies in data handling between the FeliCity API and the Digital Building Logbook (DBL). These inconsistencies were often due to inadequate validation logic, which allowed invalid data to pass through the system.

9.2 Impact Analysis

Impact analysis assesses the extent and severity of defects identified during testing, helping prioritise remediation efforts and understand their potential effects on the system's functionality and user experience.

During our previous testing phase, several high-impact defects were discovered. For instance, integration issues between the FeliCity API—specifically the FeliCity City Information Model package—and the DBL led to data inconsistencies, making the system undeployable. This caused significant delays because both packages had to be redesigned to conform to the same EF Core representation.

The recent testing phase concluded with no such severe issues.

9.3 Lessons Learned

The primary lesson learned is that testing can sometimes appear complete based on coverage metrics, while key aspects are not considered. During the previous testing phase, data modelling passed the tests because all tests used the EF Core In-Memory Database Provider. When the system was deployed to the staging environment, the database provider had to be replaced with the final one to access the MS SQL database. At that stage, previously hidden issues blocked the deployment process.

10 Recommendations

The latest round of testing concluded with some considerations.

10.1 Immediate Actions

First, we need to increase end-to-end testing coverage to support validation of user workflows. This involves developing additional test cases that cover various user scenarios and integrating these tests into our CI/CD pipeline.

10.2 Future Testing Improvements

To enhance system security, we plan to integrate automated penetration testing into our CI/CD pipeline. This will allow regular security assessments with each code commit or merge request. Tools like OWASP ZAP and Burp Suite will be configured to identify common vulnerabilities such as SQL injection and XSS.

11 Conclusion

The conclusion chapter summarises the overall findings of the testing phase and provides a final assessment of the system's readiness for deployment.

This includes an evaluation of test coverage, performance metrics, and areas for improvement identified during testing.

11.1 Summary of Findings

The EUB SuperHub services have successfully passed the internal testing conducted by FeliCity, demonstrating their functionality and reliability. Unit and integration test coverage is high in terms of line coverage, ensuring that most of the code has been executed during tests. However, branch coverage could be improved to better capture all possible execution paths and conditions within the code.

End-to-end testing coverage remains low and needs to be increased before the beta testing phase concludes.

11.2 Final Assessment

Performance testing indicates that the system performs well overall. Energy simulations are efficiently distributed, calculated, and stored, demonstrating the system's capability to handle computationally intensive tasks. The Digital Building Logbook (DBL) functions do not exhibit significant increases in response time, confirming the system's scalability and efficiency under load.

While some performance metrics on mobile devices are poor, it is important to note that the application is not primarily designed for mobile use. Given the complexity of the analysis tasks and the expected usage patterns, it is unrealistic to expect users to perform such analysis on mobile devices. Therefore, this issue is not considered to have a high impact on the system's overall performance and user experience.

11.3 Next Steps

To ensure the system is fully prepared for beta testing and eventual deployment, the following actions are recommended:

- Increase branch coverage in unit and integration tests to improve the thoroughness of the testing process.
- Expand end-to-end testing coverage to validate complete user workflows and integrate findings from these tests to address any identified issues.

To prepare for beta testing, the system should first be deployed to the staging environment. This allows for further validation in a setting similar to production, helping to catch any remaining issues. Ensuring that reference data is populated for all relevant countries is essential for the beta testing phase.

Test users should be invited and provided with detailed test protocols. These protocols must include specific tasks, expected outcomes, and reporting guidelines to allow thorough testing and valuable feedback. By addressing these areas, we enhance the system's reliability and performance, paving the way for a successful deployment and positive user reception.