

Enhancing Continuous Integration Workflows: End-to-End Testing Automation with Cypress

Maria Eduarda S. Vieira¹ ^a, Vitor Reiel M. de Lima¹ ^b, Windson Viana² ^c,
Michel S. Bonfim^{1,2} ^d and Paulo A. L. Rego² ^e

¹Federal University of Ceará, Quixadá, Ceará, Brazil

²Federal University of Ceará, Fortaleza, Ceará, Brazil

{eduardasv, vitorreiel}@alu.ufc.br, windson@virtual.ufc.br, {michelsb, pauloalr}@ufc.br

Keywords: Cypress, End-to-End, Test Automation, Quality Control, Pipeline, CI/CD.

Abstract: In Agile Software Development, adopting Continuous Integration (CI) practices enables the continuous delivery of high-quality software through frequent code deployments and automated testing. Automated tests play a crucial role in this process by reducing manual effort and increasing reliability. Among available tools, Cypress is particularly notable for executing end-to-end (E2E) tests efficiently and reliably directly within the browser environment. This paper proposes a structured approach to integrating Cypress into CI/CD pipelines, utilizing the Page Object pattern to enhance the robustness and maintainability of E2E test suites. We apply this approach in a case study in which 155 E2E tests were developed for a web-based internal system of a multinational corporation with a globally distributed user base. By detailing the methodology and results of our study, we demonstrate how this approach optimizes test execution, expands test coverage, and facilitates rapid feature deployment without compromising system stability.


1 INTRODUCTION


Agile Software Development is an iterative method known for its adaptability and efficiency in software development. In each iteration of the agile process, all essential phases of development — requirements analysis, design, coding, and testing — are addressed comprehensively (Garousi et al., 2020). This iterative approach allows for a rapid response to changes in business and project needs, ensuring continuous delivery of quality through Continuous Integration (CI) practices. With CI, code is frequently deployed to the development and/or production environment, providing a fast feedback cycle and continual improvement in the quality of delivered software.


In the current context of software development, conducting tests is an indispensable practice to ensure the quality and efficiency of systems (Pádua Paula Filho, 2019). Test automation emerges to reduce execution time and the need for manual inter-


vention, allowing tests to be executed autonomously, replicating the behavior of a real user (Mobaraya and Ali, 2019). One of the tools that stands out in this scenario is Cypress. It is a powerful tool that simplifies the process of writing, running, and debugging automated tests for web applications. This framework connects directly to the Application Under Test (AUT) from the browser, enabling fast and efficient execution of test scripts (Jyolsna and Anuar, 2022).


Unlike other tools that use specific drivers to interact with the browser, Cypress uses Document Object Model (DOM) events to send commands, resulting in faster and more reliable tests (CiteDrive, Inc., 2024). Additionally, Cypress implements automatic waiting, ensuring complete loading of DOM elements before searching for web elements, thus enhancing test robustness. Although Cypress is a powerful and widely used tool for end-to-end testing, its effectiveness greatly depends on how tests are designed and maintained. Poorly structured tests can lead to challenges such as increased maintenance complexity, duplication of test scenarios, and violations of key principles such as KISS, DRY, and SOLID (Krasnokutskaya and Krasnokutskiy, 2023; Leotta et al., 2020). Over time, these issues may result in redundant ef-

^a  <https://orcid.org/0009-0006-3846-5358>

^b  <https://orcid.org/0009-0005-9373-5536>

^c  <https://orcid.org/0000-0002-8627-0823>

^d  <https://orcid.org/0000-0001-8665-3675>

^e  <https://orcid.org/0000-0002-0936-9301>

forts, brittle test structures, and unreliable results, including false positives or negatives.

Continuous Integration and Continuous Delivery (CI/CD) is a widely adopted practice that aims to improve software quality and the efficiency of development processes up to implementation. In this method, developers frequently submit their code changes to a central repository, enabling continuous integration of new features and fixes. CI is crucial for software development quality despite initial learning challenges (Junior et al., 2023). Upon detecting new changes, the Continuous Integration tool triggers a series of automated processes known as pipelines. These pipelines perform various checks and validations on the code, providing quick feedback to developers and testers if any process fails. Tools such as GitLab, GitHub Actions, and Jenkins are widely used to implement CI/CD, which speeds up bug identification and correction and ensures that the system is always functional and ready for deployment.

Therefore, we propose an in-depth study on structuring test code using the Cypress framework, focusing on end-to-end (E2E) test automation. By combining Cypress with the Page Object Model (POM) pattern, implemented in a CI/CD pipeline, we aim to maximize the efficiency and effectiveness of automated testing processes. Our goal is to answer two research questions (RQs):

- RQ1: How does using the POM pattern in Cypress impact test maintainability and scalability?
- RQ2: What are the main challenges and best practices when using Cypress in large-scale agile projects?

This case study can benefit developers and testers by providing practical examples of efficiently structuring Cypress tests and integrating them into CI/CD pipelines while also aiding researchers by offering insights into real-world challenges and practices in applying modern tools in industrial environments, contributing to the advancement of testing methodologies. We believe that the use of Cypress in this context not only reduces test execution time but also increases test coverage and reliability, meeting the demands of modern and complex web applications.

The remainder of this paper is organized as follows. Section 2 presents the related work. The case study is described in Section 3. Section 4 discusses the experimental methods and results. Section 5 concludes the article and presents future work.

2 RELATED WORK

The interest in evaluating E2E techniques and tools is not new. For example, (Rafi et al., 2012) analyzed the benefits and limitations of software test automation, highlighting advantages like test reuse, repeatability, and increased coverage, alongside challenges such as high initial investment and the need for specific skills. Similarly, (Emery, 2009) emphasized creating keywords and reducing duplication for cost-effective tests. (Lagerstedt, 2014) demonstrated the benefits of integrating automated tests for verifying non-functional requirements, including increased productivity and fewer requirement violations. In an industrial case study, (Klammer and Ramler, 2017) described the transition to automated testing, addressing challenges in test adapter creation and result analysis but confirming its efficacy in error detection.

Concerning E2E testing tools, research has compared Selenium and Cypress in terms of performance, ease of use, and support for different browsers (Mobaraya and Ali, 2019; Bhimanapati et al., 2024). For instance, (Mobaraya and Ali, 2019) evaluated Selenium and Cypress for web application testing, highlighting Cypress's efficiency in handling dynamic elements. Similarly, (Bhimanapati et al., 2024) noted Cypress's superior speed and real-time feedback capabilities in modern web testing. The authors also pointed out Cypress's limitations in terms of browser compatibility and community size.

Some authors have emphasized the importance of quality in test construction (Krasnokutskaya and Krasnokutskiy, 2023; Leotta et al., 2020; Bicalho et al., 2024). Poorly structured tests can increase maintenance complexity and violate principles like KISS, DRY, and SOLID. (Krasnokutskaya and Krasnokutskiy, 2023) proposed patterns for Cypress test construction to improve quality and ease maintenance. In a recent study, (Bicalho et al., 2024) identified 12 common test smells in Cypress tests and assessed ChatGPT's capability to detect them, though with low precision (15%-31%). Additionally, (Jyolsna and Anuar, 2022) explored Cypress in CI/CD pipelines, demonstrating its value in E2E testing and detailed reporting.

Each study offers valuable contributions to test automation and highlights areas requiring improvement and further research. This work, however, examines a context where flexibility is essential, considering the constant improvement of processes (see 3.1) and changes of the system interface, and incorporates this consideration into the framework's development.

3 METHODOLOGY

Our case study aims to evaluate the use of Cypress in testing a real-world JSX/Java-based Web system that employs a CI/CD workflow. The system is actively maintained and continuously developed within an agile environment, leveraging automated testing to ensure high-quality software delivery. Cypress was chosen for its modern approach to end-to-end testing, offering features like real-time feedback and a simple setup process, which align with the project's fast-paced development cycle.

The main goal of our case study is to describe and analyze how Cypress is integrated into the project's testing framework and CI/CD pipeline, assessing its effectiveness in improving test coverage and identifying potential challenges in its implementation. To achieve this, we examine the structure of the test project, including the organization of test cases and adherence to best practices, while also evaluating Cypress's role in the CI/CD workflow, focusing on its impact on test automation and overall system quality.

3.1 Subject System

The target system under test is a web-based system developed for a large company that serves both academic research and industrial purposes. The web system is built on a cloud-based architecture using microservices, a relational database, and, eventually, accessing external services.

The web system can be considered medium-sized, currently in its third year of development. The project's front-end consists of a total of 813 files in various formats (JSX, JavaScript, SCSS, SVG, and JSON), with a cumulative total of 84,945 lines of code. JSX files dominate the codebase, comprising 48,214 lines of code across 481 files, highlighting their significant role in the system. On the other hand, the project's back-end is a Java Spring Boot codebase divided into 5 modules and includes a total of 612 Java classes, with 60,146 lines of code in these classes.

The project team employs agile practices for development and management, though processes are still being optimized. It comprises 24 members, including senior and junior front-end and back-end developers, designers, a project manager, a configuration manager responsible for the pipeline, and a dedicated testing team. The team also includes computer science researchers, bringing diverse expertise to the project. The team uses GitLab for versioning, automation, and deployment.

Regarding software engineering practices, there

are a lot of documents that outline functionalities from a UX/UI perspective. However, the team places significant emphasis on direct communication with the client, documenting discussions related to functionalities. These records, along with the UX/UI specifications, serve as the foundation for generating test cases, which are subsequently documented in a Google Sheets file to facilitate future automation.

3.2 Criteria for Automatization of Test Cases

As mentioned above, test cases are identified through manual testing and by examining discussions with clients and the requirements team. These test cases are documented in a Google Sheets file for tracking, referencing, and analysis for the automation process. A test case is selected for automation if (a) The system must be blocked if the test fails. For example, CRUD functions on specific elements. Or (b) It is critical that the system does not allow specific actions. For example, it is importing a file with a prohibited extension. Additionally, the automation can include component checks and validations if the clients prioritize the user interface.

3.3 Cypress Project Structure

The Cypress tests under study are allocated to a different repository from the front-end of the system, aiming to allow the test project to have its pipelines for maintenance while also being used in the front-end project for quality control, as it would be on a Test as a Service (Candea et al., 2010) project. It also means that other tests can be performed, such as integration tests (CiteDrive, Inc., 2024), without compromising the respective project. In this work, however, we focus on E2E testing.

The project has a structure based on the Page Object design pattern (Raghavendra, 2024), separating elements on levels to make the code readable and flexible. The difference between this structure and the usual structure of Page Object is that there are three layers of code: *Elements*, *Page Objects*, and *Test Cases*.

3.3.1 Elements

The Elements directory contains files that store the selectors for the components used in test cases, like buttons, inputs, labels, etc. A lot of these components are used multiple times through the tests, so it's helpful to simplify the selectors. Using JavaScript functions to return the selectors so they can be easily imported and

called in other files makes it easier to use them on different files and know their purposes.

Besides the specific Elements for each screen, there is also a General Elements file, which contains elements that can be used on most system screens, *i.e.* generic confirmation buttons or feedback toasters. Figure 1 provides a complete Elements file.

```

1 class AuditElements {
2
3   filterOption = () => {return 'label.checkbox_label'}
4   iconFilter = () => {return 'i[data-testid="test-icon-id"]'}
5   showHideFilters = () => {return 'div.audit-table-container > div > span.filter-button'}
6
7   auditResponsibleUserField = () => {return 'input[data-testid="responsible-user-dropdown"]'}
8   auditTimestampField = () => {return 'input[data-testid="timestamp-dropdown"]'}
9   auditSearchField = () => {return 'input[placeholder="Search"]'}
10  auditApplicationField = () => {return 'input[data-testid="application-dropdown"]'}
11  auditActionsField = () => {return 'div.list-container.audit-table-container > div > div > div'}
12
13 }
14
15 export default AuditElements

```

Figure 1: Cypress code: Elements file.

3.3.2 Page Objects

Inside the Page Objects directory are the files containing the methods describing the actions that Cypress will perform and the flows to complete a task. For example, deleting an Asset using the interface will take multiple clicks and actions that might generate a lot of lines of code in the test cases and even repeat themselves between test cases. The Page Objects file imports the Elements files and uses them to create a more readable and simple code, in which the team members will know exactly what every element is when editing and revisiting code. Figure 2 features a code snippet from a Page Object file, highlighting its use of both its methods and those from the General Page.

```

1 import GeneralPage from "../GeneralPage";
2 import AuditElements from "../elements/AuditElements";
3 const generalPage = new GeneralPage;
4 const auditElements = new AuditElements
5
6 class AuditPage {
7
8   before(){
9     generalPage.loginMaster()
10  }
11
12  checkMainPage(){
13    generalPage.viewAuditPage()
14  }
15
16  checkFilters(){
17    this.validateResponsibleUser()
18    this.validateTimestamp()
19    generalPage.clearAllFilters()
20    cy.get(auditElements.showHideFilters()).click()

```

Figure 2: Cypress code: Page Object snippet.

In addition to the Elements directory, there is a General Page containing generic use cases throughout the system, like opening a three-dot menu, clicking on a confirmation modal to delete elements, etc. This way, it can also be ensured that the system is consistent where it needs to be, maintaining a pattern that real users will recognize, following Jakob's Law of User Experience (Nielsen, 2017).

3.3.3 Test Cases

The Test Cases directory contains the files that Cypress will run. The .cy.js files can become lengthy and complex to read with many test cases and, by using Page Objects, we can significantly simplify these test cases. Each test case can be reduced to a single line that calls a method from the appropriate Page Object. This makes the test cases more accessible to read and maintain and streamlines the process of searching through the tests. Encapsulating the interactions with the web elements within Page Objects enhances code reusability and reduces duplication. Consequently, the overall structure of the test suite becomes cleaner, more modular, and easier to understand, leading to more efficient test management.

Figure 3 presents a code snippet from a test case file. It demonstrates how the test cases utilize methods from the designated PageObject, resulting in straightforward and concise test scripts. Additionally, it shows how users can leverage IDE shortcuts to navigate to the method contents easily.

```

1 import AuditPage from "../pageobjects/AuditPage.js"
2 const auditPage = new AuditPage
3
4 describe('Audit Interface Test', () => {
5
6   before(() => {
7     auditPage.before()
8   })
9
10  it('Validate Main Page', () => {
11    auditPage.checkMainPage()
12  })
13
14  it('Validate Filters', () => {
15    auditPage.checkFilters()
16  })
17
18  describe('Validate Search', () => {
19
20    it('Search Flow', () => {

```

Figure 3: Cypress code: Test-case File snippet.

The organization of the tests implemented in our study closely resembles the templates proposed by (Krasnokutskaya and Krasnokutskiy, 2023) for E2E test automation. For instance, the template “POM using selectors” outlines a structure similar to that of our test code, employing Locators through `cy.get()`. However, in our implementation, element Selectors are not defined directly within the `page.js` file. Instead, we utilize a separate file to store these elements as variables, which are then centrally accessed in `page.js`. We adopt about 50% of this practice in our test setup.

It is worth noting that POM is a high-level abstraction that separates web pages from test cases, enhancing code reusability. It reduces coupling between test cases and web pages, making them independent and easily reusable across different sections of the codebase. Moreover, implementing the POM simplifies test case development.

Another example of (Krasnokutskaya and Kras-

nokutskyi, 2023)’s template we used is the “*POM using locators for elements*”, which closely matches our practice of using functions named Locators. In this setup, Selectors are passed as arguments in `cy.get()`, and each Locator is retrieved via a method that returns the corresponding element. We estimate that this practice is applied in around 90% of our test setup. Lastly, the template’s recommendation to use arrow functions to simplify syntax and improve code readability is fully adopted in our implementation, with this practice being present in every test code.

3.4 Tools Necessary for Developing the Test Framework

Table 1 outlines the tools employed for Test Framework Development and describes the purpose/role of each tool and providing a quantitative overview of relevant data. Jira, GitLab, and Google Drive are used by the entire project team, not just the test team. For example, the whole team uses Jira somewhat unconventionally, primarily as an activity management tool rather than its typical focus on predefined User Stories of the functionalities. While the UX team organizes part of the user manuals on Google Docs, the test team uses Google Sheets to register test cases. A test case is composed of ID, status (On hold, Not tested, In progress, Tested, Validated, and Excluded/Invalid), scenario, description, and expected results.

Table 1: Tools employed for Test Framework Development.

Tool	Purpose	Usage Data
Cypress.io	Web test automation	Version 13.14.2, 22 spec patterns
Visual Studio Code	Integrated Development Environment	77 files, 6467 lines
GitLab	Source code management and Automation tool used for CI purpose	7 branches, 452 pipelines executed, 31 merge requests approved
Jira	Agile Project Management tool	114 issues assigned to the QA team from July 2023 to November 2024
Google Drive	Documentation	Test Cases Sheet

After installing the tools above and completing the configuration, a test folder named “Cypress” is cre-

ated, as shown in Figure 4. At this point, the Page Object Model is adopted, and tests are separated from the leading project directory to be invoked later during the deployment process in the pipeline. Within the Cypress folder structure, we use the “e2e” directory to house automated tests and the implementation of project element classes, segregated by interface and integration tests. The “fixtures” folder validates specific UI elements on certain system pages, while “support” stores generic commands used in all tests. The “videos” directory stores video recordings of failed automated tests, aiding in pinpointing where the process failed. This separation of pages and tests aims to enhance overall project organization.

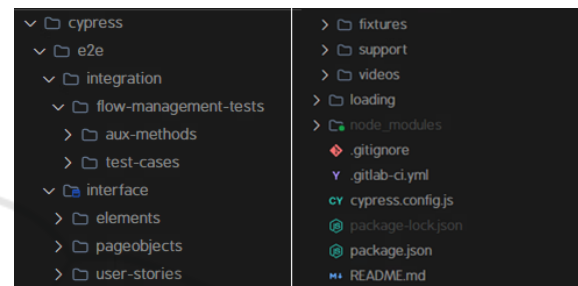


Figure 4: Folder Structure with Cypress.

Our project setup organization is based on the Cypress documentation on docs.cypress.io. They offer extensive and complete manuals on how to get started and to write the first tests.

3.5 Automated Tests on the CI/CD Workflow

Development and testing coincide within the CI process. As developers work on new functionalities within the system, they keep the testing team informed by updating the status of these implementations in Jira tasks (e.g., “In Progress”, “Ready for Review”, “Completed”). Once notified, testers create automated tests for the new system functionality by creating a local branch on their machine. They complete the development of automated tests for the new functionality and ensure that integration/interface tests pass successfully.

Figure 5 presents a pipeline configuration in GitLab CI/CD through the “gitlab-ci.yml” file, where variables control the execution of jobs. These variables include endpoints, which specify the addresses of the services or APIs to be tested, and test paths, determining whether integration or interface tests will be executed. This approach provides flexibility and reusability, allowing the same pipeline to be applied in different environments and testing scenarios with-

out requiring multiple configuration files. By centralizing the configuration through variables, pipeline maintenance is simplified, and test organization is improved, ensuring that each job executes only the necessary tests, thus optimizing time and resources.

```

30 validate-deploy:
31   stage: validate-deploy
32   extends: .install_dependencies
33   script:
34     - echo -e "\n${35};inRunning \${35};in stage with \${CI_JOB_IMAGE} ...e[0m"
35     - npx cypress cache path
36     - npx cypress cache list
37     - npx cypress verify
38     - if [ -z ${BASE_URL} ]; then echo "'BASE_URL' variable is not set. Using the default base url...";
39     - if [ -z ${GATEWAY_URL} ]; then echo "'GATEWAY_URL' variable is not set. Using the default base url";
40     - if [ -z ${FLOWSERVICE_URL} ]; then echo "'FLOWSERVICE_URL' variable is not set. Using the default";
41     - if [ -z ${AUDITSERVICE_URL} ]; then echo "'AUDITSERVICE_URL' variable is not set. Using the default";
42     - if [ -z ${USERSERVICE_URL} ]; then echo "'USERSERVICE_URL' variable is not set. Using the default";
43     - cat cypress.config.js
44     - npx run test -- --spec $(TESTS_PATH)
45   allow_failure: false
46   artifacts:
47     expire_in: 1 week
48     when: always
49     paths:
50       - cypress/screenshots
51       - cypress/videos
52   tags:
53     - linux-azure
54   rules:
55     - if: $CI_PIPELINE_SOURCE == "pipeline"
56     - if: $CI_PIPELINE_SOURCE == "merge_request_event"
57     - if: $CI_COMMIT_BRANCH
58     - if: $CI_COMMIT_TAG == /.*((Re)lease).*/

```

Figure 5: Cypress test job.

Figure 6 presents a pipeline configuration in GitLab CI/CD through the “gitlab-ci.yml” file from one of the main project repositories, where a trigger is implemented. Such trigger defines environment variables required to call the job from Figure 5. This way, the trigger passes essential information such as API endpoints and credentials. In turn, the job from Figure 5 utilizes these variables to configure and run automated tests for interface or integration.

```

407 .validate-np-deploy:
408   stage: validate-deploy
409   allow_failure: false
410   when: manual
411   inherit:
412     variables: false
413
414 validate-np-deploy-dev:
415   extends: .validate-np-deploy
416   variables:
417     BASE_URL: $BASE_URL_DEV
418     TESTS_PATH: $VALIDATE_DEPLOY_TESTS_PATH
419   trigger:
420     project:
421     branch: develop
422     strategy: depend
423     only:
424       - develop

```

Figure 6: Trigger to cypress job.

Subsequently, if all tests run without issues, a pull request is initiated with the implemented tests to the specific project test repository. Initially, a merge request is made with the Develop branch to ensure approval from the testing manager for merging into the Master branch. It’s important to note that this entire development, review, and merge request process is meticulously tracked within Jira tasks, ensuring organizational oversight and timely completion within the sprint deadline.

In the build and deployment process of end-to-end (E2E) automated tests in the project’s CI/CD pipeline, the build is initially performed in the spe-

cific test repository. After a merge into the Master (main) branch of this repository, the execution and validation process of 14 E2E automated tests are initiated. Each test targets a specific set of functionalities on different screens of the system (e.g., validating the existence of a field, manipulating a text field, uploading files/images, among others). Sometimes, the build process encounters failures, which is actually very positive. These failures are part of the quality monitoring and test maintenance process, helping to identify and resolve issues continuously.

3.6 Practices of Quality Control

We adopted the following key practices for managing automated tests effectively in our project:

- **Development First.** Considering that the test team is entirely separate from the development team, any test-first approach would be too complicated to implement. Automated tests are created after new features are developed, test cases are registered, and at least one round of manual testing is performed.
- **Wait for New Features.** It is a good practice to wait for new features to stabilize before creating automated tests and incorporating them into the pipeline. This approach prevents unnecessary rework and effort for the test team.
- **Frequent Maintenance.** Regularly updating test code in parallel with product changes ensures that tests remain effective and ready for validation. This practice allows the team to focus on minor fixes, minimizing the time and effort required to keep the test project up to date.

4 RESULTS AND DISCUSSION

4.1 Automated end to end Tests

The test team identified and implemented 155 E2E test cases for this project, divided among 12 test case files. The time to execute them in the pipeline is approximately 13 minutes.

Due to the lack of detailed documentation on requirements, these test cases were observed in manual tests and written in a separate tool. Since a private company owns the project and has not yet been launched, the authors cannot disclose specifics about these tests, which might compromise mandatory confidentiality. However, the contents of each Test-Case File can be easily described. Some steps are most common on the batteries of tests on each file, such

as: (a) Logging in the system; (b) Checking interface components based on the UX prototype; (c) Creating dependencies; (d) Creating Element; (e) Editing Element; (f) Searching and filtering Element; (g) Deleting Dependencies; (h) Deleting Element.

Figure 7 offers a detailed presentation of the file contents. It highlights that some files contain repeated test cases. Each test suite addresses the most critical aspects determined by the clients and the requirements team. For instance, File-1 thoroughly examines interface components, focusing on system navigation, while File-7 creates numerous elements to ensure that the system restricts prohibited file extensions. This pattern continues across the files.

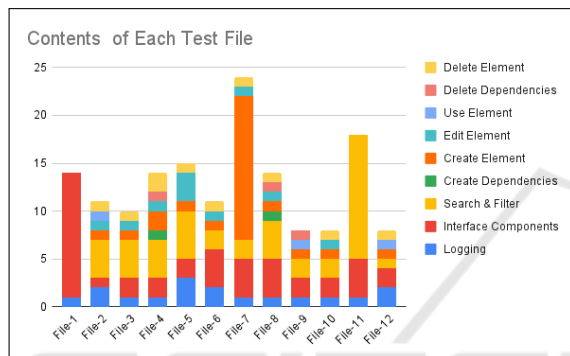


Figure 7: Contents of each test file graph.

This considered, the “Search”, “Filter” and “Delete Element” Methods are all shared between test cases from the General Page set of methods, as well as most of the methods for validation and use of Interface Components. Methods for pressing common buttons, such as Save, Cancel, and Confirm, navigating through tables, and selecting the first searched element, amongst others, and present in the General Page, are used throughout the entire test project.

4.2 Challenges and Lessons Learned

4.2.1 Wait Times

Cypress simulates human behavior based on test code specifications, making its performance reliant on application loading times. For tests that don't involve performance requirements, synchronizing wait times to ensure components fully load before tests proceed is complex and demands a deep understanding of the application and careful test code adjustments.

Another challenging aspect is to integrate automated tests into the CI/CD pipeline, which requires careful management of wait times to avoid slowing execution while ensuring good test coverage.

4.2.2 Maintenance

Maintaining automated tests is a challenging aspect of an agile environment, with constant changes to the product and, at times, to the requirements as well. The system's size and the number of business rules must also be considered. Managing dependencies is another significant challenge. The system's functionalities and elements are highly interconnected, requiring tests to run in a clean environment with no pre-existing data and without leaving any traces. This means that during a test suite run, dependencies need to be created and erased, impacting both performance and test isolation (for example, a test suite for one functionality may fail if another functionality does not work as expected within the business rules).

Due to these interdependencies, prioritizing business rules and workflows is especially difficult. As the system grows, different parts are prioritized and scrutinized, and some modules are built in stages, allowing the team to move on to new functionality before returning to complete the first. Additionally, automated tests cannot cover every possible scenario that both clients and the test team expect without becoming excessively large, redundant, and unmaintainable.

Maintaining Cypress tests also becomes challenging due to the workload burden on the testing team, especially during sprints focused on deploying new releases from development to production. As the application becomes more complex, the testing team could face increased pressure compared to other parts of the team, considering the reduced number of members. In our case, sometimes, this scenario made it difficult to keep tests up to date, particularly given other simultaneous and priority activities required by the project.

5 FINAL CONSIDERATIONS

The implementation of the proposed methodology in this case study resulted in the creation of a testing framework that effectively aligns with the project's requirements and needs while also enhancing test coverage. The study demonstrated that Cypress is not only a user-friendly and flexible tool but also integrates seamlessly with GitLab's CI/CD tools, ensuring robust and maintainable quality assurance processes. Adopting the Page Object Model design pattern proved essential, particularly in a project of this scale, by facilitating organized and scalable test automation. These advancements contributed significantly to the project's growth and continuous improvement, highlighting the benefits of integrating

Cypress into the CI/CD pipeline.

5.1 Research Questions Answered

RQ1: The POM pattern improved test maintainability by reducing code duplication and enhancing modularity. This approach made it easier to update tests when interface components changed and facilitated the scalable expansion of the test suite as new features were added.

RQ2: The key challenges identified included synchronizing wait times, managing dependencies, and maintaining tests in a dynamic agile environment. Best practices such as delaying automation until new features stabilized, performing frequent test maintenance, and leveraging dedicated pipelines in GitLab CI/CD helped overcome these challenges and ensure reliable automated testing.

5.2 Future Research Suggestions

In future work, we aim to deepen our analysis by investigating the presence of test smells and their potential impact on the effectiveness and maintainability of the testing framework. Additionally, we plan to study how the use of design patterns such as the Page Object Model and PageObject has contributed to the organization and scalability of the tests, with a focus on improving their long-term maintainability. We also intend to extend the methodology to include integration tests and explore other testing strategies, such as performance and end-to-end (E2E) testing. Finally, our goal is to refine and replicate this approach in other projects, leveraging the insights gained to establish a standardized, scalable, and efficient testing framework.

REFERENCES

- Bhimanapati, V., Goel, P., and Jain, U. (2024). Leveraging selenium and cypress for comprehensive web application testing. *Journal of Quantum Science and Technology*, 1(1):66–79.
- Bicalho, L., Montandon, J., and Valente, M. (2024). Identificação de smells em testes fim-a-fim implementados em cypress. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software*, pages 1–12, Porto Alegre, RS, Brasil. SBC.
- Candea, G., Bucur, S., and Zamfir, C. (2010). Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 155–160, New York, NY, USA. Association for Computing Machinery.
- CiteDrive, Inc. (2024). Why cypress? <https://docs.cypress.io/guides/overview/why-cypress>. Accessed: 2024-06-28.
- Emery, D. H. (2009). Writing maintainable automated acceptance tests. In *Agile Testing Workshop, Agile Development Practices*, Orlando, Florida. sn.
- Garousi, V., Keleş, A. B., Balaman, Y., and Guler, Z. O. (2020). Test automation with the gauge framework: Experience and best practices. In *Computational Science and Its Applications – ICCSA 2020*, Lecture Notes in Computer Science, 12250:458–470.
- Junior, O. d. O. B., Souza, R. H. d., and Hauck, J. C. R. (2023). Uma unidade instrucional para apoio ao ensino de integração contínua em cursos de graduação em tecnologia da informação. In *Anais do XXXIV Simpósio Brasileiro de Informática na Educação*, 10(2):378–388, Passo Fundo, RS, Brasil. SBC.
- Jyolsna, J. and Anuar, S. (2022). Modern web automation with cypress.io. In *Open International Journal of Informatics*, 10(2):182–196.
- Klammer, C. and Ramler, R. (2017). A journey from manual testing to automated test generation in an industry project. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 591–592.
- Krasnokutskaya, I. V. and Krasnokutskyi, O. S. (2023). Implementing e2e tests with cypress and page object model: evolution of approaches. In *CS&SE@ SW*, pages 101–110.
- Lagerstedt, R. (2014). Using automated tests for communicating and verifying non-functional requirements. In *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pages 26–28.
- Leotta, M., Biagiola, M., Ricca, F., Ceccato, M., and Tonella, P. (2020). A family of experiments to assess the impact of page object pattern in web test suite development. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 263–273. IEEE.
- Mobaraya, F. and Ali, S. (2019). Technical analysis of selenium and cypress as functional automation framework for modern web application testing. In *9th International Conference on Computer Science, Engineering and Applications (ICCSEA 2019)*, pages 27–46.
- Nielsen, J. (2017). Jakob’s law of internet user experience. *Nielsen Norman Group*, 18.
- Pádua Paula Filho, W. d. (2019). Engenharia de software: produtos. *LTC: Rio de Janeiro, Brazil*, v. 1.
- Rafi, D. M., Moses, K. R. K., Petersen, K., and Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42.
- Raghavendra, S. (2024). *Page Object Model (POM)*, pages 261–284. Apress, Berkeley, CA.