FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Testing Microservices within Proprietary E-Commerce SaaS: An Integration and System-Level Approach on the VTEX Platform

**Luís Filipe da Silva Jesus**


Mestrado em Engenharia de Software

Supervisor: Prof. Jorge Augusto Melegati Gonçalves

June 26, 2026

# Testing Microservices within Proprietary E-Commerce SaaS: An Integration and System-Level Approach on the VTEX Platform

**Luís Filipe da Silva Jesus**

Mestrado em Engenharia de Software

June 26, 2026

# Resumo

A adoção crescente de arquiteturas distribuídas de microserviços e plataformas *Software-as-a-Service* (SaaS) introduz desafios significativos no teste de software. Em ecossistemas SaaS proprietários, a validação de interações entre serviços, processos assíncronos e dependências externas é dificultada pela limitação da execução local, pelo estado partilhado e pelo controlo restrito sobre a infraestrutura *cloud*.

Esta dissertação aborda este problema através da conceção e avaliação de uma estratégia de testes para validar fluxos de negócio críticos. A abordagem é aplicada num caso de estudo industrial: um sistema de *e-commerce* baseado na plataforma VTEX para uma marca portuguesa de café. Ainda assim, as conclusões poderão aplicar-se a outras plataformas geridas na *cloud* onde as equipas carecem de controlo total sobre a infraestrutura e os ambientes de execução.

A estratégia combina testes de *workflows* ao nível da API com testes *end-to-end* (E2E), suportados por uma camada de abstrações para gerir o assincronismo e os dados de teste. O *Consumer-Driven Contract Testing* (CDCT) é também explorado para validar a compatibilidade de interfaces num contexto *serverless* condicionado pela plataforma.

Implementada em Cypress, a solução foi testada em cenários críticos (e.g., *checkout*, ciclo de vida de encomendas, logística e inventário). A avaliação empírica, baseada em 20 execuções independentes da *pipeline*, analisou o tempo, a estabilidade e a taxa de sucesso. Os resultados demonstram que a delegação estratégica da validação para testes de API e a gestão programática do assincronismo foram cruciais para contornar a inconstância inerente à *cloud*. Enquanto os testes E2E se mantiveram essenciais para as jornadas de utilizador, a sua elevada suscetibilidade à variabilidade do ambiente reforça a importância desta organização em camadas. É graças a estas estratégias de mitigação que a *pipeline* conseguiu superar as severas restrições da infraestrutura partilhada, garantindo um determinismo notável traduzido num tempo médio de 13,6 minutos e numa taxa de sucesso global de 90%.

Conclui-se que ecossistemas SaaS exigem uma estratégia pragmática em camadas. A validação via API, os testes E2E focados e o tratamento explícito do assincronismo fornecem, em conjunto, uma confiança significativa. Adicionalmente, verifica-se que, embora o CDCT seja conceptualmente valioso, a sua adoção prática é fortemente limitada pelas restrições de infraestrutura destas plataformas.

**Palavras-chave:** Teste de Sistema, Teste de Integração, Microserviços, Plataformas SaaS, Testes *End-to-End*, *E-commerce*, VTEX.

# Abstract

The growing reliance on distributed microservices architectures and Software-as-a-Service (SaaS) platforms introduces significant software testing challenges. In proprietary SaaS ecosystems, validating service interactions, asynchronous processes, and external dependencies is hindered by limited local execution, shared state, and restricted control over the cloud infrastructure.

This dissertation addresses these challenges by designing and evaluating a testing strategy for business-critical workflows. The approach is applied to an industrial case study: a VTEX-based e-commerce system for a Portuguese coffee retail brand. The conclusions, however, are applicable to broader cloud-managed platforms where development teams lack full control over infrastructure and execution environments.

The proposed strategy combines API-level workflow tests with selective UI end-to-end (E2E) tests, supported by a reusable helper layer to manage asynchronous behavior and test data. Consumer-Driven Contract Testing (CDCT) is also explored to validate service interface compatibility in a serverless, platform-constrained context.

Implemented using Cypress, the solution was applied to critical scenarios (e.g., checkout, order lifecycle, logistics, and inventory). An empirical evaluation based on 20 independent pipeline executions analyzed execution time, stability, and success rate. Results demonstrate that strategically delegating validation to API-level tests and programmatically managing asynchrony were crucial to bypass the inherent unpredictability of the cloud. While E2E tests remained essential for user-facing journeys, their high susceptibility to environmental variability reinforces the value of this layered organization. It is due to these mitigation strategies that the pipeline successfully overcame the severe constraints of a shared infrastructure, ensuring remarkable determinism reflected in an average execution time of 13.6 minutes and a 90% global success rate.

In conclusion, effective testing in proprietary SaaS ecosystems requires a pragmatic, layered approach. Combining API-level validation, selective E2E testing, and explicit asynchronous handling provides meaningful confidence in business-critical behavior. Furthermore, while CDCT is conceptually valuable, its practical adoption is severely limited by the infrastructure constraints imposed by these platforms.

**Keywords:** System Testing, Integration Testing, Microservices, SaaS Platforms, End-to-End Testing, E-commerce, VTEX.

# UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. They include 17 goals addressing the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, justice, economic development, and technological innovation.

Although this dissertation does not directly target environmental or social policy outcomes, it contributes indirectly to the development of more reliable, maintainable, and resilient digital systems. By proposing and evaluating a testing strategy for distributed business-critical workflows in proprietary SaaS microservices environments, this work supports software quality, automation, and the reliability of digital commerce infrastructure.

The specific Sustainable Development Goals mentioned have the following names:

**SDG 8** Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all

**SDG 9** Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

| SDG | Target | Contribution | Performance Indicators and Metrics |
|---|---|---|---|
| 8 | 8.2 | Supporting productivity improvement through technological innovation by proposing automated testing practices that reduce manual effort and improve confidence in critical workflows. | Global pipeline success rate, average execution time, and stability of automated workflow validation across repeated executions. |
| | 8.3 | Contributing to the digital capability of an industrial e-commerce context by improving the reliability of software processes that support online business operations. | Number of critical business workflows selected and validated; comparison between API-level and UI end-to-end testing layers. |
| 9 | 9.1 | Contributing to resilient digital infrastructure by addressing testing challenges in distributed, cloud-managed SaaS environments where local execution and platform control are limited. | Global pipeline success rate, average execution time, and stability of automated workflow validation across repeated executions. |
| | 9.5 | Supporting applied research and innovation in software engineering through the design and empirical evaluation of a practical testing strategy in an industrial SaaS-based case study. | Number of independent pipeline executions, evaluated business scenarios, and analysis of Consumer-Driven Contract Testing feasibility. |

# Acknowledgements

I would like to begin by thanking my FEUP supervisor, Prof. Jorge Melegati, for his guidance, availability, and valuable feedback. His insights were crucial in shaping the structure, clarity, and academic rigor of this dissertation.

I am equally grateful to my company supervisor, Hugo Carvalho, for his practical insights and constant support. His deep knowledge of the industrial context and the VTEX ecosystem was instrumental in bridging academic research with a real-world software engineering problem.

I also extend my thanks to Kodly for the opportunity to develop this project in an industrial setting. This experience allowed me to tackle a concrete, challenging problem and ground my academic work in practical application.

To my parents, thank you for making my education possible and for your unwavering support. Your continued encouragement was foundational to reaching this milestone. To my brothers, thank you for your constant companionship along the way.

To my girlfriend, thank you for your patience, support, and for being by my side during the most demanding moments of this process. Your encouragement helped me keep going when things became difficult.

Finally, I thank my friends for their companionship and for making this academic journey significantly lighter.

To all of you, thank you.

Luís Filipe da Silva Jesus

# Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.
I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

Luís Filipe da Silva Jesus

*"Everything should be made as simple as possible, but not simpler."*

Albert Einstein

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| 3PL | Third-Party Logistics |
| API | Application Programming Interface |
| CDCT | Consumer-Driven Contract Testing |
| CI/CD | Continuous Integration/Continuous Delivery |
| CRM | Customer Relationship Management |
| CSS | Cascading Style Sheets |
| DEV | Development |
| DevOps | Development and Operations |
| DOM | Document Object Model |
| E2E | End-to-End |
| ERP | Enterprise Resource Planning |
| HTML | HyperText Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| MSA | Microservices Architecture |
| OpenAPI | OpenAPI Specification |
| PRISMA | Preferred Reporting Items for Systematic Reviews and Meta-Analyses |
| QA | Quality Assurance |
| RPC | Remote Procedure Call |
| SaaS | Software as a Service |
| SFTP | Secure File Transfer Protocol |
| SWEBOK | Software Engineering Body of Knowledge |
| UI | User Interface |
| XML | Extensible Markup Language |

# Chapter 1

# Introduction

The transition to distributed microservices and Software-as-a-Service (SaaS) platforms has provided organizations with unprecedented scalability and modularity [7, 10, 20]. However, this architectural shift introduces significant complexities in ensuring overall system reliability [24, 26]. While testing isolated software components (e.g., via unit testing) often remains a straightforward and localized process, validating broader, cross-service business flows becomes considerably more challenging. Certain proprietary cloud platforms impose strict infrastructure constraints, restricting the ability to fully replicate the integrated system locally. Consequently, integration and system-level tests may need to be executed against deployed cloud environments, or against production-like preproduction deployments, in order to validate interactions that depend on the cloud platform and its surrounding services [5, 13]. In these contexts, testing approaches often struggle against unpredictable network latency, asynchronous background processes, and shared-state pollution.

This dissertation presents and evaluates a testing strategy for validating cross-service integrations and end-to-end business workflows within the constraints of the VTEX platform, a cloud-based SaaS e-commerce platform that enables organizations to build, customize, and operate online stores through a combination of managed services, APIs, and extensible applications. The approach combines API-level integration tests and end-to-end (E2E) UI (User Interface) tests to safeguard critical business processes. While the strategy is designed and assessed specifically in the context of VTEX, the challenges addressed and lessons learned may also be relevant to other proprietary SaaS ecosystems that impose similar infrastructure constraints.

Proposed as an industry dissertation by a company, the project focuses specifically on safeguarding macroscopic business workflows and assessing the feasibility and reliability of the proposed testing strategy, specifically within VTEX.

This research provides a practical, empirical and real-word solution to the challenges of quality assurance in modern e-commerce architectures, specifically within the constraints of proprietary platforms with VTEX's characteristics.

## 1.1 Motivation

The rapid evolution of e-commerce is significantly influenced by the integration of microservices and cloud computing [11]. Monolithic applications are increasingly being migrated to microservices architectures (MSA) due to the advantages they offer, such as independent development, modularity, and scalability [23, 26]. However, this architectural shift introduces complex challenges for software testing. The distributed nature of microservices leads to complex dependencies, heterogeneous execution environments, and difficulties in ensuring reliable communication between services [3]. To guarantee software quality, it is essential to balance end-to-end (E2E) testing, which assesses the entire application workflow, with API testing, which verifies the reliability of backend interactions [1].

In addition to validating complete business workflows, microservice-based systems also require mechanisms for ensuring that independently evolving services remain compatible at their interfaces. In distributed architectures, failures may arise not only from incorrect internal logic, but also from mismatches between the requests a consumer sends and the responses a provider actually returns. Consumer-Driven Contract Testing (CDCT) is therefore relevant as a complementary strategy, since it aims to capture and verify the expectations that consumers have of provider services before incompatibilities propagate to broader integration or end-to-end tests [16, 22, 27]. However, its practical applicability in proprietary SaaS environments is not straightforward, particularly when provider services depend on serverless runtimes, platform-managed state, or cloud-only execution environments. This motivates the need to examine whether CDCT can provide practical value in the constrained VTEX context, alongside API-level and UI-based workflow validation.

Testing macroscopic flows becomes even more challenging when integrating these systems within proprietary SaaS platforms. Unlike traditional environments where the entire infrastructure can be fully controlled and replicated, modern SaaS often involves ready-made software that requires testing strategies specifically adapted to cloud dependencies [24]. While unit tests can effectively cover custom business logic in isolation, they are insufficient for validating interactions with the platform's native triggers and cross-system data flows. In ecosystems like VTEX, developers operate under a "cloud-only" delivery model where core components, proprietary serverless runtimes, and native data layers cannot be fully instantiated locally.

This theoretical challenge is compounded by practical industry needs. This dissertation stems from a concrete requirement proposed by Kodly, a technology consultancy and software development company with its headquarters in Porto, specialized in digital transformation, B2C and B2B eCommerce. The company developed, maintains, and evolves a VTEX store for a major Portuguese brand in the coffee sector.

Ensuring that custom macroscopic workflows — such as the checkout process, Customer Relationship Management (CRM) synchronization, and third-party logistics (3PL) integrations — function seamlessly is a critical business imperative for the company. Failures in these broad flows directly impact revenue and customer experience. Consequently, the automated tests de-

signed to safeguard these comprehensive journeys must be executed directly against live cloud environments. This lack of local execution introduces severe testing impediments, including network dependency, unpredictable asynchronous behavior from background processing pipelines, and limited environment control that can lead to state leakage across tests. Addressing these cloud-imposed limitations requires an adapted, pragmatic testing strategy that moves beyond traditional methods to ensure that critical e-commerce business flows remain reliable and deterministic in a real-world industrial context.

## 1.2 Research Questions

Motivated by the necessity to validate complex business workflows in a constrained cloud infrastructure for a live e-commerce operation, this dissertation seeks to answer the following overarching research question:

*How can an effective integration/system-level testing strategy be designed and implemented for microservices within a proprietary e-commerce SaaS platform like VTEX?*

To address this main question comprehensively, the research is further broken down into three specific research questions:

- **RQ1:** *How can automated testing be engineered to mitigate the inherent constraints of cloud-only microservices environments?*
- **RQ2:** *What are the quantitative trade-offs in execution time, standard deviation (of execution time) and test stability when validating macroscopic business flows via API-level integration tests compared to UI end-to-end tests?*
- **RQ3:** *To what extent is CDCT viable and beneficial for ensuring interface compatibility in a serverless proprietary cloud ecosystem where local infrastructure control is highly restricted?*

## 1.3 Objectives

The primary objective of this dissertation is to design, implement, and evaluate a testing strategy for validating business-critical workflows in proprietary SaaS environments, using a VTEX project as an industrial e-commerce case study. While the solution is implemented and assessed within the constraints of a real VTEX-based e-commerce system, the underlying principles are intended to inform similar contexts where development teams must validate distributed business behavior under limited control over the platform infrastructure.

To accomplish this, the following specific objectives were defined:

- To characterize the main testing challenges imposed by proprietary cloud-managed SaaS platforms, particularly regarding limited local execution, asynchronous behavior, external dependencies, and persistent shared state.
- To define and select high-value business flows and integration points within the e-commerce system that are critical for integration and system-level validation, ensuring that the testing strategy focuses on scenarios with significant business impact.
- To design and implement a multilayered automated test suite that combines API-level workflow tests with selective UI-based end-to-end tests.
- To develop programmatic abstractions, in the form of a reusable helper layer, to support workflow triggering, asynchronous state observation, recursive polling, and API-driven cleanup.
- To empirically compare the execution efficiency and reliability of API-level and UI end-to-end testing layers by analyzing metrics such as execution time, success rate, standard deviation (of execution time), and framework intervention frequency (to assess test stability).
- To explore the viability of CDCT, evaluating its architectural alignment and practical feasibility within a serverless, proprietary cloud context.

## 1.4 Dissertation Structure

The remainder of this dissertation is structured as follows:

- Chapter 2 (State of the Art): Reviews the current literature regarding testing in microservices architectures. It explores the challenges and methodologies associated with end-to-end testing, API testing, and consumer-driven contract testing, while addressing the specificities of cloud and SaaS environments.
- Chapter 3 (Problem Characterization): Characterizes the problem addressed by this dissertation, focusing on the validation of complete business workflows in proprietary, cloud-managed SaaS platforms. It identifies the main constraints affecting this type of validation, including network dependency, asynchronous processing, limited environment control, external dependencies, and slower feedback loops.
- Chapter 4 (Proposed Testing Strategy): Presents the proposed testing strategy for validating business-critical workflows in proprietary SaaS environments. It describes the strategy's principles, applicability assumptions, layered validation approach, handling of asynchronous behavior, state management concerns, contract testing as a complementary mechanism, and CI/CD integration.

- Chapter 5 (Case Study: Application to the VTEX Platform): Applies the proposed strategy to a real-world VTEX-based e-commerce system. It introduces the VTEX platform, describes the system architecture and integrations, and details the practical implementation of the automated test suite, including Cypress-based API and UI E2E tests, helper abstractions, contract testing evaluation, and pipeline automation.
- Chapter 6 (Results and Discussion): Presents and analyzes the data collected from the automated test executions. It evaluates global pipeline metrics, compares API-level and UI end-to-end execution behavior, and discusses the impact of asynchrony and retries on test stability.
- Chapter 7 (Conclusion): Synthesizes the main findings and contributions of this dissertation, answers the research questions, discusses limitations, and proposes directions for future work.

# Chapter 2

# State of the Art

This chapter presents a systematic literature review conducted to characterize the state of the art in testing strategies for microservice-based and distributed systems. Although the review considers the general landscape of testing strategies, its focus is on approaches that are relevant to the validation of macroscopic business flows in complex, multi-service systems.

The objective of the review is to identify existing testing strategies, techniques, and tools that can inform the design of a testing approach for proprietary SaaS environments. In particular, the review examines general testing levels and techniques in microservice-based systems, followed by a more focused analysis of E2E testing, API testing, and Contract Testing. These strategies are discussed in terms of their benefits, limitations, applicability, and relevance to the validation of cross-service business behavior.

The chapter also considers specialized contexts and non-functional aspects related to cloud and SaaS environments, since these factors influence how testing strategies can be applied in practice.

## 2.1 Microservices Context

### 2.1.1 Microservices Architecture

Microservices architecture (MSA) is an architectural style for building applications as collections of small, independent services, each centered on a specific business capability. Unlike monolithic systems, where all modules are tightly coupled, microservices operate in their own processes and communicate through lightweight protocols, such as HTTP APIs, remote procedure calls (RPC), or message queues. This loose coupling allows services to be developed, deployed, updated, and scaled independently, improving system flexibility, maintainability, and operational efficiency [7, 10, 20].

Ponce et al. [20] further emphasize that microservices facilitate continuous delivery, automated deployment, and containerization, making them particularly suitable for complex, dynamic domains like e-commerce. By enabling modularization around business functions and decoupled interactions, microservices provide an architecture that supports rapid adaptation to changing requirements while maintaining reliable end-to-end business processes.

### 2.1.2 Challenges in Microservice Testing

Ibrahim and Luong [11] state that, despite their benefits, microservice-based architectures "introduce complexities related to inter-service communication, data consistency, and debugging." While this architectural shift enables faster and more independent deployments, it also introduces substantial challenges for Quality Assurance (QA) [26]. In distributed environments, a single business process often spans multiple services, databases, and third-party integrations. Ensuring system reliability therefore requires not only validating the logic of individual components, but also guaranteeing the correctness of the interactions between them.

These challenges are particularly evident in e-commerce systems built on microservice architectures. Although individual services can be developed, deployed, and scaled independently, core business functionality — especially under peak-load conditions such as Black Friday — relies on coordinated interactions across multiple services and demands end-to-end reliability. As a result, comprehensive testing must extend beyond isolated components to validate system-level behavior and cross-service interactions [26]. Santos and Martinho [21] present a case study on the integration of e-commerce and Enterprise Resource Planning (ERP) systems, illustrating the practical challenges that arise in managing complex platform operations. Such examples highlight the need for testing approaches that ensure correct system-wide behavior rather than focusing solely on individual components.

## 2.2 Methodology

In order to identify relevant literature on testing strategies for microservice-based systems, a systematic literature review was conducted. search was performed using the Scopus and IEEE Xplore digital libraries. A query was construted using keywords related to this work's main topics and was iteratively refined to balance comprehensiveness and relevance. The final search query was used in both databases, using each one's default search fields:

```
(microservices OR microservice architecture OR
microservice-based systems)

AND

(software testing OR end-to-end testing OR integration
testing OR system testing OR API testing OR contract
testing OR consumer-driven contract testing)
```

To complement the database search, backward snowballing was applied to selected publications. This process resulted in a total of 385 records, of which 377 were identified through database searches and 8 through other sources. After removing duplicate records, 339 unique studies remained for screening. The study selection process is summarized in Figure 2.1.

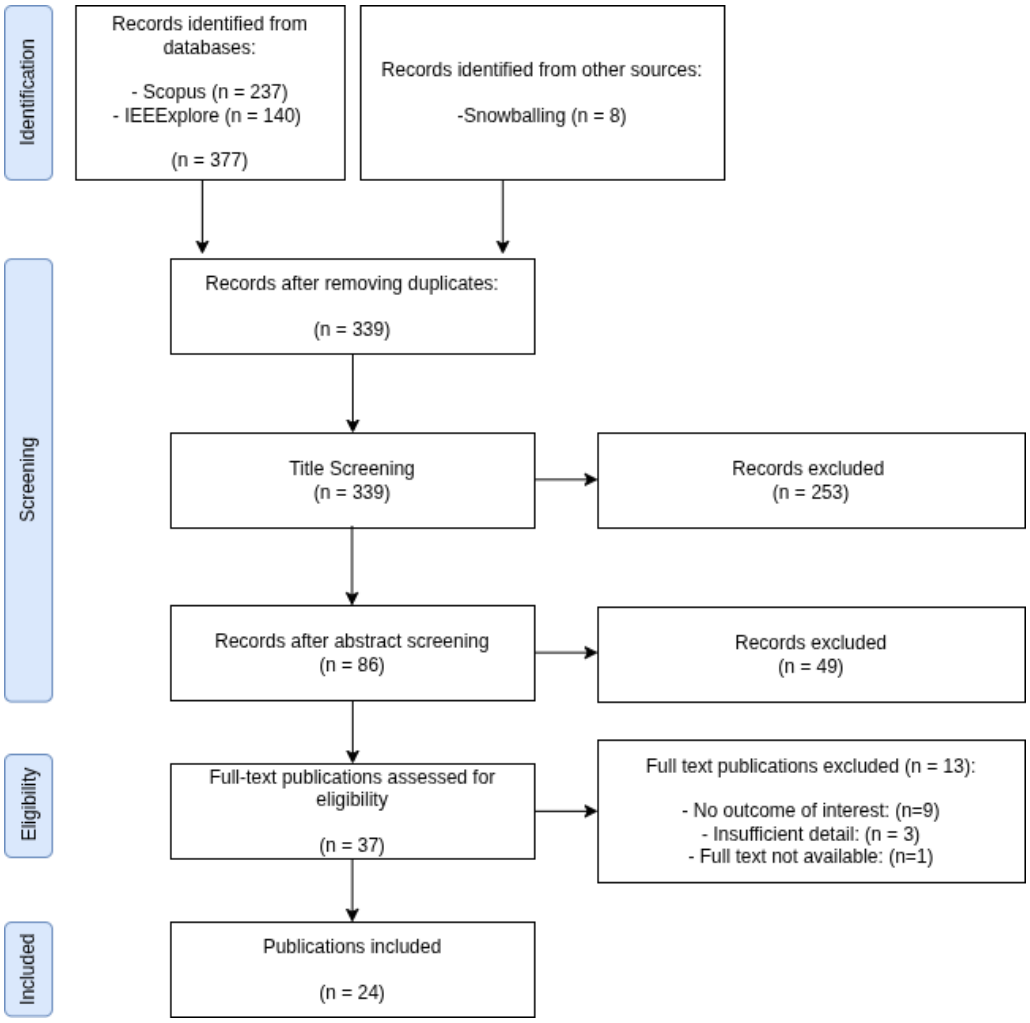



Figure 2.1: PRISMA flow diagram of the publications selection process.

### 2.2.1 Study Selection

The selection of studies was conducted in multiple stages, following a progressive screening approach.

#### 2.2.1.1 Title Screening

During the title screening phase, records were evaluated to exclude publications that were clearly outside the scope of this review. Titles were assessed based on their relevance to microservice-based or distributed software architectures, software testing, e-commerce systems, or software architecture in complex platforms. Studies were excluded at this stage if their titles indicated no relevance to any of these areas. To minimize the risk of premature exclusion, a conservative approach was adopted, retaining any records where the title suggested potential relevance.

As a result of this screening, 253 records were excluded, and 86 publications proceeded to abstract screening.

#### 2.2.1.2 Abstract Screening

The abstract screening phase aimed to assess the relevance of studies at a conceptual level, in line with the defined search strategy. Abstracts were evaluated to determine whether the study addressed microservice-based or distributed systems within a software engineering context, considered testing or system-level quality assurance concerns, or provided architectural or domain insights relevant to complex e-commerce or large-scale software platforms. In case none of these aspects were clearly addressed in the abstract, the publication was excluded from further consideration.

Publications were excluded if their abstracts demonstrated a focus on unrelated domains, systems of insufficient complexity, or a lack of relevance to both testing strategies and contextual architectural challenges. Following this phase, 49 additional records were excluded, leaving 37 publications for full-text assessment.

#### 2.2.1.3 Full-Text Eligibility Assessment

The full texts of the remaining publications were assessed using detailed inclusion and exclusion criteria. At this stage, studies were evaluated based on their contribution to testing strategies for microservice-based systems, testing tools, as well as their relevance in providing contextual understanding of the challenges inherent in microservices architectures or e-commerce.

Thirteen studies were excluded during full-text assessment for the following reasons: no relevant outcome or contribution to the objectives of this review (n = 9), insufficient technical detail or analytical depth (n = 3), and unavailability of the full text (n = 1).

### 2.2.2 Included Studies

Following the completion of the selection process, 24 studies were included in the final analysis. These studies constitute the basis for the state-of-the-art discussion presented in this work, encompassing both testing-focused research and contextual studies that inform the challenges and constraints of testing microservice-based e-commerce systems.

## 2.3 Testing in Microservices-based Applications

The Software Engineering Body of Knowledge (SWEBOK) defines four fundamental testing levels, each addressing a different scope of validation [12]. Unit testing focuses on verifying individual components in isolation, while integration testing evaluates interactions between combined units to detect interface and interoperability issues. System testing assesses the behavior of the fully integrated system against functional and non-functional requirements, such as performance and security. Finally, acceptance testing validates whether the system satisfies user needs and is ready for operational deployment.

Beyond testing levels, SWEBOK also categorizes test techniques according to how test cases are derived. Specification-based (black-box) techniques design tests from requirements and functional specifications, while structure-based (white-box) techniques rely on knowledge of the internal implementation. Other approaches include experience-based techniques, which leverage tester expertise, as well as usage-based and fault-based techniques that focus on operational behavior and fault detection capabilities [12]. This classification is particularly relevant in microservices-based systems, where independently developed and deployed services often lack full visibility into internal implementations.

### 2.3.1 The State of Research on Microservices Testing

Ponce et al. [20] conducted a systematic literature review of microservices testing, identifying 74 primary studies and classifying them according to the SWEBOK framework. Their results show that system testing is the most frequently investigated level in microservice-based systems, followed by integration, unit, and acceptance testing. Functional testing objectives such as conformance, regression, and API testing dominate the literature, while non-functional concerns mainly address performance efficiency and reliability. The authors highlight gaps in non-functional testing, but functional testing and particularly system-level validation—despite its high prevalence regarding the most frequently investigated testing levels — still require further research to address the specific challenges of each system context. Additionally, the authors observe that most approaches rely on specification-based testing strategies and are evaluated through laboratory experiments with low-to-medium technology readiness levels, indicating that microservices testing research remains at an early stage of maturity.

Moreover, Ponce et al. [20] discuss several research efforts related to test generation and optimization techniques in microservice-based systems, such as test prioritization and automated test case generation. However, these approaches are often tailored to specific use cases, system configurations, or testing objectives, which limits their general applicability across heterogeneous microservice ecosystems. The review also highlights works that employ end-to-end tests as part of regression testing strategies, acknowledging their relevance for validating cross-service behavior. Additionally, this study mentions contract-based validation, namely for early detection of issues in the development lifecycle, but does not go deep into the topic and its relevance for validating service interactions. On the other hand, Miao, Shaafi, and Song [16] provide a more thorough review on contract testing and mention their critical role in microservices testing and how they may strengthen or be a complementary strategy to other testing techniques:

> "[...] behavior verification and API-level contracts often share overlapping goals in guaranteeing consistency across service boundaries." [16]

However, even though contract testing is said to be "widely practiced" [16], the fact is that considering the literature covered in this review, it is still an underexplored topic in the academic context, with few works addressing implementation details, best practices and empirical evaluations of its effectiveness in real-world scenarios.

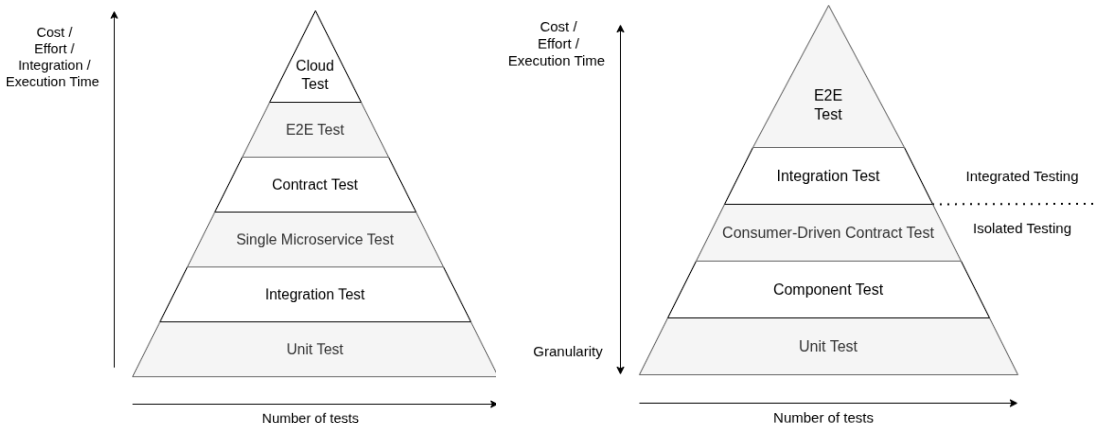
Abdelfattah et al. [1] propose an automated approach for computing metrics to assess the completeness of end-to-end and API test suites. While promising as a means of quantifying test coverage at the system level (in terms of covered microservice endpoints), this solution remains a proof of concept and requires further extensibility to support additional programming languages and testing frameworks. Given these characteristics, and although coverage metrics constitutes an important research direction, this work deliberately does not focus on proposing or evaluating such techniques. Instead, the emphasis is placed on understanding and analyzing testing strategies that provide confidence in system-level behavior and service interactions.

Nevertheless, this study still provides relevant insights that highlight the importance of complementing end-to-end testing with API tests, stating that:

> "E2E testing [...] assesses the entire application workflow [...], and API testing [is] focused on verifying the reliability of the application's backend through direct interactions with its APIs. Striking a balance between these approaches is essential for achieving comprehensive test coverage." [1]

This observation aligns with the idea that testing a system, and particularly more complex ones, requires a multi-layered approach that combines different testing strategies.

In a broader system-oriented perspective, Bello-Trejo et al. [3] conduct a systematic literature review on testing strategies for microservice-based systems. Their findings confirm that unit, integration, and end-to-end testing are the most frequently applied strategies in both academic and industrial contexts, while contract testing is acknowledged as a relevant approach but remains comparatively less explored in depth. The review also highlights that, although end-to-end testing is widely used to validate system behavior, it is associated with significant challenges related to cost, test fragility, and environment management. These observations reinforce the need for complementary strategies that enable earlier and more focused validation of service interactions, particularly at service boundaries.




(a) Li et al. (2020). (b) Lehvä et al. (2019).

Figure 2.2: Testing pyramids comparison.

Another relevant contribution is presented by Li et al. [15], who propose a system-oriented testing strategy for microservice-based applications based on an extended testing pyramid. Their model shows a testing pyramid (Figure 2.2a) where end-to-end tests and contract tests are explicitly considered, which we do not commonly find in the literature. The authors advocate for integrating contract testing and end-to-end testing alongside unit, integration, and single-microservice tests. As illustrated in their testing strategy model, tests closer to the business level (e.g., end-to-end tests) are fewer in number and higher in cost, while lower-level tests provide faster feedback and greater isolation.

This conceptual model is particularly relevant to the context of this work, as it explicitly positions contract testing as a mechanism for validating service boundaries — i.e., the interfaces and interaction contracts between independently developed services — while end-to-end testing is used to validate complete business flows.

A similar conceptual model outlining testing levels for microservice architectures, that explicitly show where end-to-end and contract testing fit in the overall testing strategy for microservices, is also mentioned by Gong and Cai [8]. The relative positioning of each testing level is very similar to the one presented by Li et al. [15] in Figure 2.2a. Moreover, the authors also discuss the challenges of end-to-end testing, and also provide more in-depth discussion on contract testing, that is further explored in the next sections.

While both Li et al. [15] and Gong and Cai [8] place contract testing above integration testing, Lehvä, Mäkitalo, and Mikkonen [14] place it below (Figure 2.2b). This lack of consensus highlights an ambiguity in the literature, reflecting different interpretations of testing levels in microservice-based systems [2]. At the same time, it underscores how different contexts may lead to different perspectives on the role of each testing level or technique. Contract testing can be considered a higher-level testing strategy, as it necessarily focuses on validating agreements between independent services, while integration testing, depending on its scope, does not necessarily involve two microservices, but for example, a microservice and a database, that may be seen as a single unit (part of the same system). On the other hand, following the logic of Lehvä, Mäkitalo, and Mikkonen [14], one could argue that in terms of granularity and scope, contract testing is more granular and focused since it is specifically targeted to the interface between services — validating that they adhere to their defined contracts — whereas integration testing is more concerned to the behavior of combined services as a whole, belonging to a broader system context and, consequently, a higher level in the pyramid.

To ensure that microservice-based systems function correctly from the user perspective through to backend processing, two complementary approaches are commonly employed in practice and research: end-to-end testing, which validates complete user workflows across service boundaries, and API testing, which provides more targeted verification of backend interfaces and data exchange. These approaches work together to provide confidence in system behavior.

#### 2.3.1.1 End-to-End Testing

End-to-end (E2E) testing is a key approach for validating the overall behavior of software systems, covering everything from user interactions to backend processing and integrated services. Unlike tests that focus on individual components or their interfaces, E2E testing examines the system as a whole to ensure it delivers the intended functionality to users. As applications increasingly rely on complex front-end frameworks and distributed back-end architectures, traditional testing methods alone are often insufficient to capture potential issues across the full workflow [28].

Despite their importance, these tests are among the most challenging to design, implement and maintain in order to successfully validate the desired flows [8, 17]. Moreover, the implementation of an E2E test suite must not only consider the tests themselves, but also adhere to best practices such as the integration with CI/CD pipelines, that sometimes may pose a significant challenge [8, 28].

Ngo et al. [17] also highlight that "developers tend to limit the number of tests they perform on the system level due to the high risk of flakiness and also the costs of running tests on the whole system every time". This is consistent with the assertion by Li et al. [15], that reinforces the importance of choosing the test scope carefully, especially when an external service or interface is expected to cause uncertainty, recommending the exclusion of such tests or uncertainties from an E2E test suite, delegating their validation to other forms of testing.

Considering these challenges, this dissertation takes them into account when defining the proposed testing strategy. In particular, E2E testing is not treated as a testing layer to be applied indiscriminately to all scenarios, but as a mechanism that should be reserved for the most relevant and cost-effective business workflows [10]. This principle later informs the selection of test cases and the balance between browser-based E2E tests and API-level validation.

Tools for E2E testing are widely available. Hernández et al. [9] provide a study on tools that are specifically designed for microservices, and end up selecting two tools (Jaeger and Zipkin) for a more in-depth comparison. The study reveals that Jaeger outperforms Zipkin in terms of execution and failure detection. However, while this study is very focused on microservices, the tools may not be very suitable for our context, since there is not as much freedom in terms of deployment and instrumentation of the services, given that they are integrated in a third-party SaaS platform (VTEX). Thus, end-to-end testing would be applied in a living development environment, where the services are already deployed and running, and not in a controlled testing environment.

Considering this, more traditional tools that are also used in monolithic applications such as Cypress, Selenium, Playwright and Puppeteer may be more suitable for the context of this work and are also acknowledged as relevant tools [6, 8]. García et al. [6] compare these tools and generally conclude that Cypress and Playwright are the ones providing the best features and performance. Nevertheless, the choice of tool must also consider other factors such as the proximity to the development stack.

#### 2.3.1.2 API Testing

API testing is more "focused on verifying the reliability of the application's backend through direct interactions with its APIs" [1].

Since these tests directly interact with specific endpoints, they are generally faster and cover endpoints and interactions that may not be triggered or exercised by end-to-end tests, which makes them a valuable complement to E2E testing:

> "E2E tests interact with the system through the user interface, [...] while API tests interact through direct individual endpoint calls or composite calls that include multiple such endpoint calls." [1]

Thus, since API tests can be more targeted and also chained to cover specific workflows, they can actually be used to test full business flows as well, with a more focused scope on the underlying backend logic that the developer is interested in validating.

A significant challenge is maintaining the sequence of API calls, which highlights the importance of a proper test design and organization [1].

Regarding tools for API testing, as long as there is support for HTTP requests, implementing these tests as a complement to traditional E2E tests should be straightforward and even integrated in the same testing framework. For instance, Cypress also provides support for API testing, which makes it a good candidate for implementing both E2E and API tests.

### 2.3.2 Ensuring Interface Compatibility: Contract Testing

While end-to-end and API testing validate system behavior and backend functionality, ensuring compatibility between independently evolving services requires additional focus on their interfaces. This leads to contract testing as a specialized technique for interface validation.

In microservice architectures, services rely on lightweight APIs to communicate and exchange data. This presupposes that services send and receive data in specific formats and structures, which can be understood as an agreement that both parties must respect for communication to be successful. In this context, contract testing emerges as a strategy aimed at ensuring that services conform to predefined interaction contracts [16, 27].

Schwarz, Quast, and Riehle [22] frame this problem in terms of syntactic interoperability, noting that, unlike traditional compilation errors, API incompatibilities typically surface only at runtime. This limitation is exacerbated in microservice-based systems due to polyglot technology stacks and independent service deployments, which make maintaining API compatibility increasingly challenging. Consumer-driven contract testing has been proposed as a means to address this issue through isolated verification of service interfaces, while complementing existing testing strategies rather than replacing them.

Rather than exercising multiple services together, contract testing evaluates compatibility by comparing a provider's behavior against interaction expectations defined by its consumers. This

enables the detection of interface mismatches independently of full system integration and supports the early identification of incompatible service versions during the development lifecycle [14, 23]. When both parties adhere to the defined contract, it can be used as a shared basis to verify each side of the integration [14].

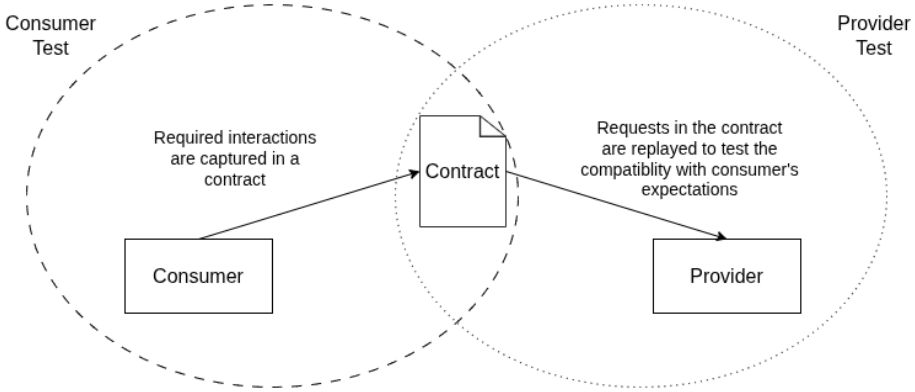



Figure 2.3: Consumer-Driven Contract Testing (CDCT) overview.

A recent survey on contract testing in microservice architectures [23] highlights that, despite increasing industrial adoption, academic research on the topic remains relatively limited. From a large initial corpus of studies, only a small subset explicitly addresses contract testing in microservice-based systems, with most publications emerging after 2019. The survey identifies several recurring research themes, including the use of contract testing to support microservice interface design, service evolution, and quality assurance, as well as its role in reducing the complexity of testing highly distributed systems. The authors also report that consumer-driven contract testing tools, such as Pact and Pact Broker [18], are the most commonly referenced solutions in the literature.

Lehvä, Mäkitalo, and Mikkonen [14] present a case study on the application of CDCT in a microservice-based web store composed of eight services and four databases, which initially relied solely on unit, component, and end-to-end tests. While the authors recommend complementing contract testing with additional testing approaches, they provide detailed insights into the practical implementation of CDCT and report notable benefits from its adoption. In particular, the case study indicates improvements in early defect detection and a reduction in flakiness and non-determinism associated with end-to-end tests.

Ayas et al. [2] also conduct an empirical analysis focused on four open-source microservice systems that adopt consumer-driven contract testing, proposing a testing architecture that positions CDCT tests alongside unit, integration, component, and system tests. The authors also underline the importance of CDCT as a more explicit structure for inter-service integration.

In summary, contract testing provides a focused means of validating service interactions at their boundaries, thereby complementing broader testing strategies such as end-to-end and API testing. However, the studies presented usually assume that there is full control over the services and infrastructure being tested, which may not be true in our context due to the SaaS nature of the e-commerce platform in which this work is framed, that may pose technical limitations. These

open questions suggest that deeper exploration of contract testing strategies, tool support, and best practices is both necessary and promising for improving integration reliability and guiding practical adoption in real-world systems.

### 2.3.3 Specialized Contexts and Non-Functional Aspects

Beyond the core functional testing strategies, several works address specialized testing contexts and non-functional aspects that are relevant to microservice-based systems.

#### 2.3.3.1 Testing in Cloud/SaaS Environments

Sotiriadis et al. [24] discuss the testing of modular cloud services. This is particularly relevant to the nature of this work, since the environment in which the proposed testing strategy is applied is an e-commerce application built on VTEX, which is a cloud-based SaaS platform. This may pose an even greater challenge, since "traditional applications are firstly designed and then tested, however, today SaaS involves on-the-shelf ready-made software [...] [that] requires testing plan preparation according to the cloud enablers and their inter-dependencies" [24].

This interdependency between cloud services also affects the environment in which testing can be performed. Kiran and Simons [13] note that, when the service under test collaborates with other services, testing may be conducted either using simulated stubs or mock services, or using a context of live web services. This distinction is important for SaaS-based systems, because some integration behavior depends not only on the service implementation itself, but also on the deployed services and external dependencies with which it interacts. Therefore, although mocks and stubs remain useful for isolated validation, some integration tests may need to exercise a realistic or live service environment in order to provide confidence in cross-service behavior.

Sotiriadis et al. [24] propose a structured methodology combining unit and integration testing through both top-down, specification-driven, and bottom-up, use-case-driven approaches. While this methodology addresses the challenge of testing service compositions and their interactions, it remains largely focused on API-level validation and process-oriented integration testing. In particular, it does not explicitly address the validation of service boundaries (interfaces through which independently developed services interact), nor does it consider end-to-end testing as a means to validate complete business flows across independently evolving services.

The methodology also provides a structured way of identifying and organizing test cases based on service interfaces and use-case decomposition; however, this specification is largely formal and documentation-driven, and less suited to capturing explicit behavioral expectations in highly dynamic service-based systems.

#### 2.3.3.2 Performance and Reliability Testing

In addition to functional testing strategies, there are also approaches that focus on the non-functional behavior of microservice-based systems, particularly regarding performance and reliability. Camilli et al. [4] propose MIPaRT, a methodology and platform for integrated performance and reliability

testing of microservices within DevOps pipelines, combining ex-vivo workload generation with monitoring data to derive performance and reliability metrics at the level of individual request classes and services, and emphasizing the role of DevOps practices as a fundamental enabler of continuous microservice testing. Although MIPaRT primarily targets extra-functional qualities rather than functional correctness, its system-level, DevOps-oriented perspective on testing complements the focus of this work by reinforcing the need to align testing strategies — including end-to-end and contract testing — with the operational realities of microservice-based e-commerce platforms.

#### 2.3.3.3 Automated Test Generation

Giamattei et al. [7] evaluate several specification-based black-box testing techniques and introduce a combinatorial strategy, uTest, which automatically derives functional and robustness test cases from OpenAPI specifications, thereby reducing the manual effort of writing tests, even though automated test generation itself is not the main focus of this work. Crucially for the present study, the authors argue that "microservices have probably better not be tested independently, but considering the entire architecture they are part of" which further reinforces the importance of end-to-end, system-level verification of microservice-based systems and aligns with the emphasis placed here on validating complete business flows and cross-service interactions. Moreover, the study emphasizes the usefulness of API specifications (e.g., OpenAPI) as a source of truth for deriving test cases.

### 2.3.4 Synthesis and Gap Identification

Overall, even though most of the works mentioned acknowledge various techniques and testing levels, there is a clear consensus on the importance of validating the system as a whole, particularly in microservice-based applications where business processes often span multiple independently developed services. This underlines the relevance of defining a testing strategy that not only covers end-to-end business flows but also ensures the correctness of interactions between services [3, 8, 15, 20]. In this context, end-to-end testing emerges as a critical approach for verifying complete business workflows, while API testing provides a complementary layer of coverage, capturing interactions that may not be exercised by traditional end-to-end tests. Contract testing further supports this strategy by validating service boundaries and ensuring that independently developed services adhere to their expected interfaces.

The literature converges on the critical need for a multi-faceted testing strategy in microservices that combines:

1. Validation of business flows through end-to-end testing to ensure correct user journeys.
2. Targeted backend verification through API testing to complement end-to-end coverage.
3. Interface compatibility assurance through contract testing to ensure stable service interactions and enable independent deployment.

4. Integration with DevOps practices to manage the cost and complexity of these tests in continuous delivery environments.

However, a clear gap exists between the proposed conceptual models and their empirical application in constrained, real-world environments. Most studies, including those on contract testing, assume full control over the service infrastructure. This assumption may not hold in SaaS-based e-commerce ecosystems (like VTEX [25]), where development teams operate within the technical and operational constraints of a third-party cloud platform.

Therefore, this work aims to address these gaps by exploring the practical implementation of a testing pipeline that follows the best practices identified in this review. The focus is on applying end-to-end and API testing, as well as evaluating the feasibility and effectiveness of contract testing within this specific context, as part of a comprehensive strategy for microservice-based e-commerce applications.

## 2.4 Conclusion

Testing in microservice-based applications, particularly in the context of e-commerce platforms, remains a complex and multifaceted challenge. The literature consistently emphasizes the need to validate not only individual services but also the interactions and workflows that span multiple, independently developed services — an issue especially critical in systems where failures can directly impact revenue, customer experience, and operational continuity.

Unit, integration, and end-to-end testing are widely applied and serve complementary purposes, yet each comes with limitations. Unit and integration tests offer rapid feedback and localized verification but fail to capture system-level behavior. End-to-end tests provide comprehensive validation of business processes but are costly, fragile, and difficult to maintain, particularly in distributed or SaaS-based environments. API testing addresses some of these gaps by offering targeted verification of backend interactions, allowing validation of workflows that may not be exercised through the user interface.

Within this landscape, contract testing has emerged as a particularly valuable strategy for ensuring the correctness of service interactions at their boundaries. CDCT provides a structured approach to validate that independently developed services adhere to agreed-upon interaction contracts. Empirical studies and case reports demonstrate that CDCT can improve early defect detection, reduce the flakiness of end-to-end tests, and support interface evolution in microservices [2, 22, 27]. However, literature indicates that contract testing is still underexplored in terms of implementation details and guidance, best practices, and evaluation in real-world scenarios. Existing studies often assume full control over service infrastructure — a condition that may not hold in SaaS-based platforms — introducing additional constraints and open questions.

Taken together, the reviewed literature highlights gaps and opportunities for future research. There is a need for empirical studies investigating contract testing and its integration with other strategies — unit, API, and end-to-end — in operational environments, including those with limited control over service infrastructure. Exploring these aspects can provide actionable insights

into optimizing testing strategies for microservices, improving integration reliability, and ensuring robust end-to-end business flows in complex e-commerce platforms. In this context, further investigation into tool support, best practices, and adaptation to diverse technological stacks is clearly necessary, laying the groundwork for more effective testing methodologies in large-scale, business-critical microservice-based applications.

# Chapter 3

# Problem Characterization

This chapter characterizes the problem addressed by this dissertation. While Chapter 1 introduced the motivation and research questions, and Chapter 2 reviewed the relevant literature on testing in microservice-based and SaaS environments, this chapter narrows the discussion to the specific type of validation considered in this work: the testing of complete business workflows in proprietary, cloud-managed SaaS platforms.

The focus is not on software testing in general, nor on the validation of isolated units of code. Instead, the problem concerns integration and system-level validation of business-critical scenarios that may span user-facing components, platform-managed services, custom backend logic, asynchronous processing mechanisms, and external systems. In such contexts, a business workflow may be exercised through the user interface, through APIs, or through a combination of both, depending on which part of the system behavior is being validated.

The chapter first discusses the nature of complete business workflow validation in distributed SaaS systems. It then describes the constraints introduced by proprietary, cloud-managed platforms and explains why these constraints make conventional testing assumptions difficult to apply.

## 3.1 Validation of Complete Business Workflows

In microservices systems, a single business scenario involves multiple layers and systems, including user-facing applications, platform APIs, custom backend services, managed data repositories, event processing mechanisms, and external enterprise systems such as CRM, logistics, invoicing, or payment providers.

In this context, validating isolated components is not sufficient to provide confidence that the business process works correctly as a whole. For example, a checkout interaction in an e-commerce application may succeed from the user's perspective but still fail to trigger the expected downstream processing. Similarly, a backend integration may behave correctly in isolation but fail when executed as part of a broader workflow involving platform events, persistent data, and external dependencies.

The problem addressed in this dissertation therefore concerns the validation of business workflows across integration boundaries. These workflows may include visible user-facing steps, such as browsing products or completing checkout, but they may also include backend processes that occur after the visible interaction has finished. Examples include customer synchronization, order processing, logistics dispatch, inventory synchronization, and invoice generation.

This distinction is important because end-to-end validation does not necessarily mean that every scenario must be exercised exclusively through the browser. Some scenarios require UI-level validation because the user interaction itself is part of the behavior under test. Others can be validated more directly through APIs when the relevant behavior concerns backend processing, integration logic, or cross-system data propagation.

## 3.2 Cloud-Managed SaaS Constraints

The validation of complete business workflows becomes more challenging when the system is built on top of a proprietary SaaS platform. In traditional self-hosted microservices systems, development teams can often reproduce substantial parts of the infrastructure locally or in controlled test environments. Services can be executed in containers, databases can be reset between test runs, and dependencies can be replaced with mocks or test doubles when isolation is required.

In proprietary SaaS environments, this level of control may be unavailable. The platform core, serverless runtime, native data layers, event processing mechanisms, and managed APIs are operated by the platform vendor and cannot be fully instantiated locally by the development team. As a result, many relevant behaviors can only be validated after custom components have been deployed to a cloud environment managed by the platform.

This limitation changes the nature of integration and system-level testing. Instead of executing tests against local services under full control, the test suite must interact with deployed cloud environments. Even when these environments are not production environments, they still depend on real platform services, persistent data stores, network communication, asynchronous processing, and sometimes external systems.

For the validation of complete business workflows, this constraint is particularly significant. If the objective is to validate behavior that depends on platform-managed services, replacing those services with local mocks may reduce the value of the test. Mocking can be useful for unit or component testing, but it cannot fully reproduce platform-specific behavior such as native triggers, checkout processing, asynchronous event propagation, or data synchronization across external systems. Consequently, the testing problem must be understood in relation to the limited control that development teams have over the runtime environment.

## 3.3 Main Difficulties in Workflow Validation

Running integration and system-level tests against cloud-managed environments introduces several difficulties. These difficulties do not necessarily originate from defects in the application

under test, but they affect the reliability, repeatability, and execution cost of the validation process. This section summarizes the main sources of difficulty that characterize the problem.

### 3.3.1 Network Dependency

Since the tests interact with cloud-hosted services, they depend on network availability, latency, and stability. Even when the application logic is correct, temporary network delays or transient communication failures may affect the execution of requests and responses. This can increase execution time variability and may lead to failures that are not directly caused by the business logic being validated.

This issue becomes more relevant as the scope of the tested scenario increases. A complete business workflow may require several interactions across different services and systems. Each additional network boundary increases the potential for latency, instability, or delayed responses. As a result, tests that validate broader workflows tend to be more exposed to environmental variability than tests that execute isolated logic locally.

### 3.3.2 Asynchronous Processing

Many business workflows in SaaS-based systems are not executed synchronously from beginning to end. Instead, they may depend on background processing, platform triggers, event handlers, scheduled jobs, or external system responses.

This creates a gap between the moment when a workflow is triggered and the moment when its final state becomes observable. A test that checks the expected result too early may fail even though the system would eventually reach the correct state. At the same time, the duration of these background processes may vary across executions, making it difficult to define deterministic validation points.

Asynchronous behavior therefore makes workflow validation more complex than immediate request-response testing. The absence of an expected state at a given instant does not necessarily indicate failure; it may simply indicate that the system has not yet completed its internal processing.

### 3.3.3 Limited Environment Control

Another important difficulty is the limited ability to control or reset the test environment. In local or self-managed systems, it is often possible to recreate databases, roll back transactions, restart containers, or isolate each test execution in a clean environment. In proprietary SaaS platforms, these mechanisms are often unavailable or only partially available.

Platform-managed data stores are persistent and may be shared across development workspaces, tenants, or services. Data created during one test execution may remain available in later executions, affecting assumptions made by subsequent tests. This increases the risk of state leakage and makes tests more difficult to repeat reliably.

This problem is especially relevant when validating realistic business scenarios, because such tests often create or modify persistent business entities such as customers, orders, addresses, invoices, or inventory records. If the environment cannot be reset completely, previous executions may influence future ones, reducing the determinism of the test suite.

### 3.3.4 External Dependencies

Complete business workflows often involve systems outside the direct control of the development team. In e-commerce platforms, these may include CRM systems, logistics providers, invoicing services, payment gateways, or other enterprise integrations. These systems may have their own availability constraints, response times, test environments, data models, and release cycles.

External dependencies increase the difficulty of validation because failures may originate outside the custom codebase. A test may fail because an external system is unavailable, because test data is inconsistent between systems, or because an integration partner changes behavior unexpectedly. At the same time, excluding these dependencies entirely may reduce the realism of the test when the objective is to validate an integrated business process.

This creates a tension between isolation and realism. More isolated tests tend to be faster and more deterministic, but they may fail to capture issues that only appear when the real integration is exercised. More realistic tests provide higher confidence in deployed behavior, but they are slower and more exposed to instability outside the direct control of the development team.

### 3.3.5 Slower Feedback Loops

Cloud-managed platforms also introduce slower feedback loops. In local development, developers can modify code and immediately execute tests against local services. In proprietary SaaS environments, relevant behavior may only become testable after deployment to a cloud workspace or tenant. This means that validation depends not only on test execution time, but also on deployment time, environment availability, and platform processing latency.

This limitation has practical consequences for the design of the test suite. If every relevant scenario requires a broad, slow, and fragile test, then the feedback provided to developers becomes less useful. On the other hand, if the tests are too narrow, they may fail to validate the business workflows that actually matter from an operational perspective. The problem is therefore not simply to maximize coverage, but to determine how to validate critical workflows with an acceptable balance between confidence, reliability, and execution cost.

## 3.4 Implications for the Testing Approach

The difficulties described in this chapter show that validating complete business workflows in proprietary SaaS environments shows a different set of challenges than validating fully controlled systems. The relevant behavior often emerges from the interaction between custom components, platform-managed services, asynchronous mechanisms, persistent data, and external systems. At

the same time, the development team has limited control over the infrastructure in which those interactions occur.

Table 3.1 summarizes the main aspects that characterize this validation problem. The table is not intended to define the proposed solution, but to consolidate the reasons why complete business workflow validation becomes difficult in proprietary cloud-managed SaaS environments.

| **Problem aspect** | **How it affects workflow validation** |
|---|---|
| Complete business workflows | Relevant behavior may span user-facing components, platform services, custom backend logic, asynchronous processing, and external enterprise systems. |
| Cloud-managed platform constraints | Some platform-dependent behavior cannot be fully reproduced locally, requiring validation against deployed cloud environments. |
| Asynchronous and delayed processing | Expected outcomes may only become observable after background processing, platform triggers, scheduled jobs, or external system responses. |
| Limited control over state and dependencies | The development team may not be able to fully isolate executions, reset persistent platform state, or control external systems involved in the workflow. |
| Execution variability and feedback cost | Network dependency, deployment requirements, delayed observability, and broad workflow scope can increase execution time and reduce test stability. |

Table 3.1: Summary of the problem aspects affecting complete business workflow validation in proprietary cloud-managed SaaS environments.

Together, these aspects create a testing problem with competing concerns. Ideally, integration and system-level tests would be executed in environments that are both realistic and highly controlled, allowing developers to reproduce workflows without depending on a full cloud deployment cycle. However, in proprietary SaaS platforms, relevant behavior may depend on platform-managed services, native APIs, event mechanisms, and persistent data layers that cannot be fully instantiated locally by the development team. As a result, some business workflows must be validated against deployed cloud environments, even when this reduces isolation, makes state harder to control, delays the observability of outcomes, and increases execution variability.

The next chapter addresses these implications by proposing a testing strategy for this class of problem. The strategy is then applied and evaluated in the VTEX case study presented in the following chapters.

# Chapter 4

# Proposed Testing Strategy

This chapter proposes a testing strategy for validating complete business workflows in proprietary, cloud-managed SaaS environments. The strategy is designed in response to the problem characterized in Chapter 3, where meaningful validation requires interaction with deployed platform behavior, while the development team has limited control over infrastructure, state reset, asynchronous processing, and external dependencies.

The proposed strategy is not intended to replace all existing testing practices. Unit and component tests remain valuable for validating isolated business logic where such isolation is possible. However, the focus of this chapter is on integration and system-level validation of business-critical scenarios that cross architectural and organizational boundaries. The strategy is therefore aimed at cases where confidence depends on exercising realistic workflows involving platform services, custom components, persistent data, asynchronous processes, and external systems.

The core idea is to combine realistic cloud-based execution with selective test scope. Instead of relying exclusively on broad user-interface end-to-end tests, the strategy separates validation concerns across different execution layers. API-level tests are used whenever they can validate the relevant business behavior more directly and reliably, while UI-based end-to-end tests are reserved for scenarios where the user-facing interaction itself is part of the behavior under test. This is supported by reusable test abstractions for data preparation, workflow triggering, state observation, and cleanup.

## 4.1 Strategy Overview

The proposed strategy combines realistic cloud-based execution with selective test scope. Its purpose is to support the validation of business-critical workflows in environments where the platform cannot be fully reproduced locally and where important behavior depends on managed services, persistent data, asynchronous processing, and external integrations.

A central aspect of the strategy is the use of a realistic non-production cloud environment. Since relevant platform behavior may only be observable after deployment, integration and system-level tests must be able to execute against an environment where the managed platform services

are available. At the same time, this environment must be isolated from production data and real user activity, as the validation of business workflows may create or modify persistent business entities.

The strategy also separates validation concerns across different execution layers. The assignment of a scenario to a layer is driven by a single criterion: whether the user-facing interaction is itself the real concern under validation. If it is, a UI end-to-end test is used. If the relevant behavior can be triggered and observed through backend interfaces alone, an API-level workflow test is preferred. Applying both layers to the same scenario should remain the exception, reserved for cases where each test validates a genuinely distinct concern — the UI test focusing on the user journey, and the API test focusing on backend correctness independently.

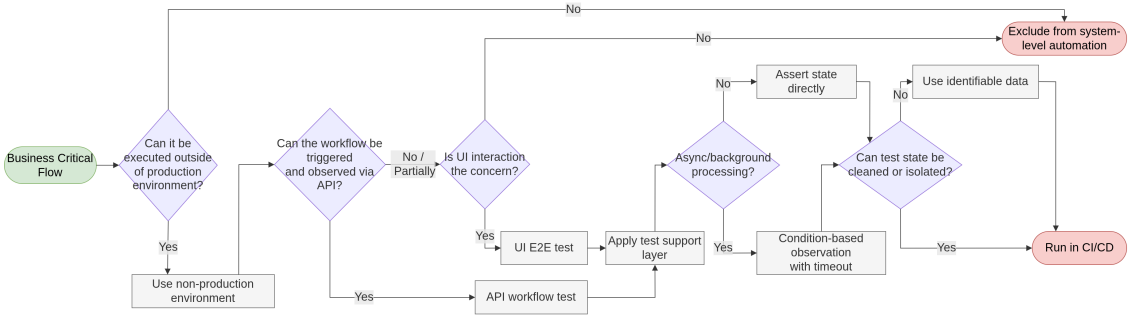



Figure 4.1: Decision workflow for selecting and designing system-level automated tests.

Asynchronous behavior is treated as an expected characteristic of the environment rather than as an exceptional condition. In cloud-managed microservices systems, state changes may become visible only after platform triggers, background jobs, or external systems have completed their processing. The proposed strategy therefore requires tests to observe business states explicitly and to account for delayed propagation when validating expected outcomes.

Repeatability is another fundamental concern. Since proprietary SaaS environments often do not provide full reset capabilities, tests must be designed to minimize state leakage between executions. This requires controlled test data, stable identifiers, reusable setup and cleanup mechanisms, and support abstractions that encapsulate low-level interactions with the platform and external systems.

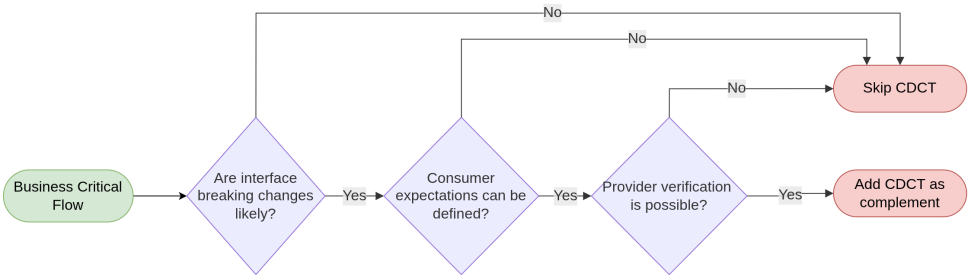



Figure 4.2: Decision workflow for evaluating the applicability of Consumer-Driven Contract Testing (CDCT).

Contract testing is also considered as a complementary validation mechanism when the main risk is interface compatibility between services. Unlike API-level workflow tests and UI end-to-end tests, contract tests are not intended to validate a complete business workflow. Instead, they can provide additional confidence that independently evolving services respect the expected interaction structure, provided that consumer expectations can be defined and provider verification can be executed.

Overall, the strategy does not attempt to eliminate the constraints of proprietary SaaS environments. Instead, it adapts the design of integration and system-level tests to those constraints, combining realistic execution with careful scope selection, explicit state observation, and reusable test support mechanisms. Figures 4.1 and 4.2 provide an overview of the proposed strategy. Figure 4.1 summarizes the main decision process for selecting and designing system-level automated tests for a business-critical workflow. Figure 4.2 presents the complementary assessment used to determine whether Consumer-Driven Contract Testing (CDCT) is applicable and should be implemented as a complement. The following sections describe each element of the strategy in greater detail.

## 4.2 Applicability and Assumptions

The proposed strategy is intended for systems built on top of proprietary or managed SaaS platforms whose core services cannot be fully executed locally by the development team. In these contexts, relevant behavior, especially at integration and system level, often depends on deployed cloud infrastructure and cannot be validated through purely local execution.

The strategy is particularly applicable when business workflows span multiple components or services. These may include user-facing applications, platform APIs, custom backend services, managed data repositories, asynchronous event mechanisms, and external enterprise systems. The approach assumes that the relevant business behavior emerges from the interaction between these elements, rather than from isolated components alone.

The strategy also assumes a hybrid control model. The development team can implement and test custom components, but does not control the full platform runtime or all external dependencies. Some parts of the system can therefore be engineered directly, while others must be treated as externally managed infrastructure whose behavior can only be observed through available interfaces.

For the strategy to be practical, the platform or surrounding system should expose APIs that can be used for test execution, state observation, or data management. These interfaces are important because they allow tests to validate backend workflows, prepare data, trigger relevant operations, and inspect resulting states without relying exclusively on browser automation.

Finally, the strategy assumes the existence of a non-production environment where tests can be executed without affecting real users or production data. Since the validation of realistic business workflows may create or modify persistent entities, executing such tests directly against production would introduce unacceptable operational risk in most scenarios.

These assumptions delimit the scope of the proposal. The strategy is not intended as a universal approach for all forms of microservice testing. Instead, it addresses the specific problem of validating business-critical workflows in environments where complete local control is unavailable but realistic cloud-based validation remains necessary.

## 4.3 Environment

A central requirement of the proposed strategy is the existence of an isolated non-production environment. Since integration and system-level tests may create, modify, and delete business data, they should not execute against production systems. Instead, tests should run in a dedicated development, staging, or test tenant configured as similarly as possible to production.

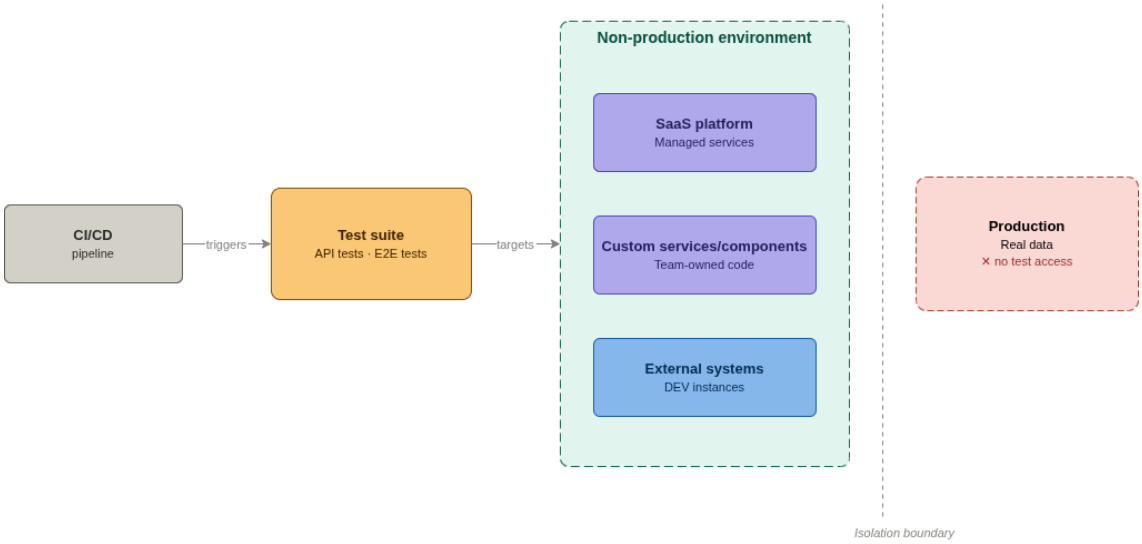



Figure 4.3: Environment example for executing integration and system-level tests in proprietary SaaS environments.

Figure 4.3 illustrates this principle. The automated test suite is executed from the CI/CD pipeline and targets a non-production environment that contains the relevant SaaS platform services, custom components, and non-production external dependencies. Production remains outside the test execution boundary, preventing automated tests from affecting real customers or business data.

In practice, this isolation may be implemented differently depending on the SaaS platform. For example, a platform may provide separate accounts, tenants, workspaces, or staging environments. In the case study later presented in this dissertation, this principle is instantiated by using a separate VTEX account for development and testing, configured to resemble the production account as closely as possible. The external enterprise systems involved in the tested workflows also provided development instances, allowing the tests to exercise realistic integrations without interacting with production data.

This environment should satisfy two goals that are often in tension. On one hand, it must be isolated enough to prevent tests from corrupting real business data or interfering with real users. On the other hand, it must remain realistic enough to exercise the relevant platform behavior and integrations. Excessive simplification of the environment may make tests faster and more deterministic, but it can also remove the very behavior that the tests are meant to validate.

External dependencies should follow the same principle. Whenever possible, the non-production SaaS environment should be connected to non-production instances or sandbox environments of external systems. This allows tests to validate realistic integration behavior while reducing operational risk. If such environments are not available, the strategy must explicitly assess whether the dependency can be safely mocked, whether the scenario should be excluded from automated system-level validation, or whether the test should be limited to a narrower integration boundary.

The environment strategy therefore does not prescribe a single infrastructure model. Instead, it establishes a principle: tests should execute in the most realistic environment that can be used safely and repeatedly. The case study presented in Chapter 5 instantiates this principle using the environment mechanisms available in the selected SaaS platform.

## 4.4 Layered Workflow Validation

The proposed strategy adopts a layered approach to workflow validation. Rather than treating UI-based end-to-end tests as the default mechanism for validating all business scenarios, each scenario is assigned to the execution layer that best matches the behavior being evaluated.

### 4.4.1 API Tests

API-level workflow tests are used to validate business behavior that can be exercised through backend interfaces without relying on user-interface interaction. These tests may involve a sequence of API calls that prepare data, trigger a workflow, and verify the resulting state in one or more systems.

This type of test is particularly useful when the relevant behavior concerns backend processing, integration logic, asynchronous propagation, or communication with external systems. By avoiding browser rendering and UI interaction, API-level tests tend to provide faster and more stable feedback than UI end-to-end tests, while still validating meaningful cross-service behavior.

In the proposed strategy, API-level tests are not limited to isolated endpoint checks. They may validate complete business scenarios, as long as the scenario can be triggered and observed through available interfaces. For example, a test may create a business entity, trigger a processing operation, and verify that the expected result becomes visible in another system. The important criterion is not whether the test uses a browser, but whether it validates the relevant business outcome across the appropriate boundaries.

### 4.4.2 UI E2E Tests

UI end-to-end tests remain necessary when the user-facing interaction is itself part of the behavior under test. This includes scenarios where the objective is to validate that a user can complete a critical journey, that the frontend integrates correctly with platform services, or that important interface state transitions occur as expected.

However, UI end-to-end tests should be used selectively. In cloud-managed SaaS environments, browser-based tests are exposed not only to backend and network variability, but also to rendering delays, third-party scripts, dynamic DOM (Document Object Model) structures, and frontend asynchronous loading. As a result, using UI tests as the main validation mechanism for all business logic can make the test suite slow, fragile, and difficult to maintain.

The proposed strategy therefore reserves UI end-to-end tests for critical user-facing journeys and relies on API-level workflow tests for scenarios where backend behavior can be validated more directly. This separation improves maintainability while preserving confidence in the most important user interactions.

### 4.4.3 Role of Unit Tests

Although this dissertation focuses on integration and system-level validation, unit and component tests still have an important role. They should be used whenever the behavior under test can be isolated without losing the meaning of the validation. Examples include pure business logic, data transformations, utility functions, and deterministic validation rules within custom components.

However, unit tests cannot validate behavior that depends on platform-managed services, native triggers, persistent cloud data, or external system propagation. For this reason, they are considered complementary to the proposed strategy rather than sufficient for the problem addressed in this work.

## 4.5 Test Support Layer

To make cloud-based workflow validation maintainable, the proposed strategy introduces a reusable test support layer. This layer encapsulates recurring operations that would otherwise be duplicated across tests, such as creating test data, triggering workflows, querying system state, waiting for asynchronous results, and cleaning up generated entities.

The purpose of this layer is not only code reuse. In proprietary SaaS environments, the support layer also acts as a buffer between test scenarios and platform-specific operational details. Instead of embedding low-level API calls, credentials, identifiers, waiting logic, and cleanup procedures directly inside each test, these concerns are centralized in helper functions or service abstractions.

This improves the test suite in three ways.

First, it improves readability. Test scenarios can be written in terms of business actions and expected outcomes rather than low-level technical operations.

Second, it improves maintainability. If an API endpoint, authentication mechanism, or data format changes, the update can be applied in one support function rather than across many tests.

Third, it improves reliability. By centralizing synchronization and cleanup logic, the support layer reduces the risk that each test implements these concerns differently and inconsistently.

In the proposed strategy, the support layer should include at least four types of operations: setup operations, workflow-triggering operations, observation operations, and cleanup operations. Setup operations create or configure the initial state required by a scenario. Workflow-triggering operations initiate the business process under test. Observation operations retrieve the state used for assertions. Cleanup operations remove or neutralize test data after execution whenever possible.

## 4.6 Handling Asynchronous Behavior

Asynchronous processing is one of the main sources of instability in cloud-managed workflow validation. Since relevant state changes may depend on background jobs, event handlers, platform triggers, or external systems, tests cannot assume that the expected result will be available immediately after the triggering action.

The proposed strategy addresses this by using condition-based observation. Instead of relying on fixed waiting times, tests should repeatedly observe the relevant system state until the expected condition is satisfied or a maximum timeout is reached. This approach allows tests to continue as soon as the system reaches the expected state, while still failing when the expected outcome does not occur within an acceptable time window.

Condition-based observation is preferable to static waits for two reasons. First, static waits make tests slower than necessary when the system completes the operation quickly. Second, they remain unreliable when the operation occasionally takes longer than expected. A fixed delay is therefore both inefficient and brittle. By contrast, condition-based observation adapts to the actual processing time of the environment.

The expected condition should be defined in business terms whenever possible. For example, instead of waiting for a fixed duration after triggering a workflow, the test should wait until the corresponding business entity reaches the expected state, until a record becomes available in a target system, or until a specific integration result can be observed.

This mechanism should also be bounded. Infinite waiting would hide failures and block the pipeline. Each asynchronous validation should therefore define a timeout that reflects the expected behavior of the system and the acceptable delay for that scenario.

## 4.7 State Management and Repeatability

Because proprietary SaaS environments often do not provide full environment reset capabilities, repeatability must be explicitly engineered into the test strategy. Tests should be designed so that repeated executions do not depend on hidden state left by previous runs.

Controlled and identifiable test data is essential in this context. Entities created during test execution should include predictable identifiers, prefixes, metadata, or other markers that distinguish them from real or manually created data. This makes it easier to locate, validate, and clean test artifacts.

Tests should also avoid unnecessary dependence on pre-existing mutable state. Whenever possible, each scenario should create the data it requires or verify that the necessary preconditions are present before execution. This reduces the probability that changes in the environment invalidate the assumptions of the test.

Cleanup is equally important when the platform exposes safe mechanisms for doing so. Depending on the system, this may involve deleting entities, reverting fields, cancelling records, or otherwise neutralizing the side effects of the test. In some cases, complete cleanup may not be possible because certain business records are immutable or because deletion is not exposed by the platform. In those situations, the test should at least minimize the impact of generated data and ensure that subsequent executions can distinguish their own artifacts.

State management is therefore not treated as a secondary implementation detail. In cloud-managed environments, it is a core aspect of test design. Without explicit state management, integration and system-level tests become increasingly unreliable as test data accumulates over time.

## 4.8 Contract Testing as a Complementary Strategy

Contract testing is included in the proposed strategy as a complementary mechanism for validating service interactions. While API-level and end-to-end tests exercise broader business workflows, contract testing focuses specifically on whether communicating services respect the expected structure and behavior of their interfaces.

In CDCT, the consumer defines the interactions it expects from a provider, including the request format and the expected response structure. The provider can then be verified against those expectations. This makes contract testing particularly useful when the main risk is interface incompatibility between services that evolve independently.

Within the proposed strategy, contract testing should be adopted whenever it provides sufficient confidence for validating a service boundary without requiring the execution of a broader workflow. In such cases, it can reduce the need for some costly end-to-end validations and provide earlier feedback about incompatible API changes.

However, contract testing is not treated as a replacement for integration or system-level tests. It validates whether services agree on their interaction contracts, but it does not necessarily validate the complete business behavior that emerges from a deployed workflow involving platform services, persistent data, asynchronous processing, and external systems. For this reason, contract testing is most useful when combined with the other validation layers of the strategy.

The proposed approach therefore considers contract testing as an optional but valuable layer. When the required consumer and provider conditions are available, it should be implemented to

strengthen interface compatibility checks. When those conditions are not sufficient, broader API-level or end-to-end workflow tests remain necessary to validate the behavior of the integrated system.

## 4.9 CI/CD Integration

The proposed strategy should be integrated into the development and deployment workflow. Since proprietary SaaS systems often require deployment before meaningful validation can occur, automated tests should run as part of the pipeline against the selected non-production cloud environment.

The purpose of CI/CD integration is to make validation repeatable and operationally useful. If tests are executed only manually, they are less likely to be run consistently and may fail to detect regressions before changes are promoted to more stable environments. By integrating the test suite into the pipeline, the team can obtain feedback after deployment and before release.

However, the pipeline must account for the cost and variability of cloud-based tests. Not every test necessarily needs to run on every code change. The strategy may distinguish between faster API-level validations that run more frequently and broader UI end-to-end validations that run at selected stages, such as before promotion to a staging or production-like environment.

Secrets, credentials, and environment configuration must also be managed carefully. Since these tests interact with real cloud services and possibly external systems, authentication data should be stored securely in the CI/CD platform and should not be hardcoded in the test repository.

CI/CD integration therefore operationalizes the testing strategy. It ensures that the proposed approach is not only executable by developers locally or manually, but also embedded into the regular release process.

## 4.10 Summary of the Proposed Strategy

Table 4.1 summarizes the proposed strategy and relates each element to the problem it addresses.

The proposed strategy addresses the problem characterized in Chapter 3 by combining realistic cloud-based validation with controlled test scope. It does not attempt to eliminate the constraints of proprietary SaaS environments but adapts test design to those constraints by selecting appropriate execution layers, explicitly handling asynchronous behavior, managing persistent state, and embedding validation into the development pipeline.

Table 4.1: Mapping between the problem aspects identified in Chapter 3 and the proposed testing strategy.

| **Problem aspect** | **Proposed strategy** |
|---|---|
| Complete business workflows | Validate each workflow at the most appropriate abstraction level by combining API-level integration tests with selective UI end-to-end tests. |
| Cloud-managed platform constraints | Execute the automated test suite against a dedicated non-production cloud environment. |
| Asynchronous and delayed processing | Use explicit synchronization through state observation, recursive polling, and bounded timeouts to detect when workflows have completed. |
| Limited control over state and dependencies | Employ controlled test data, explicit setup and cleanup mechanisms, and isolated external dependencies whenever supported by the platform. |
| Execution variability and feedback cost | Reduce browser automation to critical user-facing journeys, centralize reusable testing utilities, and integrate the suite into CI/CD to improve execution efficiency and repeatability. |

# Chapter 5

# Case Study: Application to the VTEX Platform

This chapter presents the concrete industrial case study used to apply the proposed testing strategy: a real VTEX-based e-commerce store for a Portuguese coffee retail brand. It introduces the VTEX platform, focusing on the platform characteristics that are relevant to testing and workflow validation. The chapter then describes the main custom components and integrations that define this system under test, highlighting the architectural boundaries and execution models that affect the validation of business-critical workflows. Finally, it presents how the proposed strategy was implemented in this case study, including the automated test suite, supporting abstractions, and practical validation choices adopted in this context.

## 5.1 Overview of the VTEX Platform

### 5.1.1 Platform Architecture

The project was developed in the context of an e-commerce platform built on VTEX, a cloud-based SaaS solution widely used in digital retail.

VTEX [25] follows a microservices-based architecture and provides a set of built-in modules that cover common e-commerce functionalities, such as catalog management, checkout, logistics, payments, and pricing. These modules are managed by the platform and communicate internally, allowing standard e-commerce flows to work without requiring custom infrastructure setup.

From a developer perspective, the platform can be seen as composed of three main parts: the core modules, the VTEX IO development environment, and the Master Data repository. Despite these core modules, the platform is flexible and does not impose the utilization of all of them in every project. For example, it is possible to use another data storage solution (for data that is independent of the core modules) running on a separate infrastructure instead of Master Data, as long as the necessary integrations are implemented.

To support custom business requirements, VTEX provides VTEX IO, a serverless development environment. It allows teams to build custom frontend applications and backend services

that extend the platform's default behavior. These services typically communicate with VTEX Core through HTTP APIs, often using pre-built clients and abstractions provided by the platform libraries.

Data is managed through VTEX Master Data, which acts as a central repository for storing business information, such as customer data or application-specific entities. It follows a document-based model and is also responsible for triggering integrations with external systems through configurable events.

### 5.1.2 Accounts and Workspaces Constraints

The platform is organized around accounts and workspaces. An account represents a store and its configuration, while workspaces work similarly to development branches, allowing multiple versions of the system to coexist. Workspaces can be used for parallel development, but they do not always provide full isolation. In particular, some resources — such as Master Data — are shared across workspaces within the same account. This means that data created during testing in one workspace may be visible in another, potentially leading to conflicts and data pollution.

The following sections describe how each system component amplifies or interacts with these constraints.

## 5.2 System Architecture and Integrations

This project extends the VTEX platform through several custom components and integrations with external systems. At a high level, the system includes a custom storefront, a custom checkout UI, a CRM Middleware service, and a backend service responsible for logistics, invoicing, and inventory synchronization (Logistics/Invoicing Service). There are more services (such as a custom tax calculation service), but these are the foundational ones that represent the main architectural boundaries and integration points relevant to the testing strategy.

These components interact both with VTEX native modules and with external enterprise systems, resulting in a distributed architecture with multiple integration points. Figure 5.1 presents a simplified layered view of the architecture developed on top of the VTEX platform. The purpose of the diagram is not to provide a formal model or an exhaustive representation of all interactions, but rather to identify the main architectural boundaries that are relevant to the testing strategy.

The diagram distinguishes between customer-facing components, VTEX Core services, VTEX IO custom services, Master Data, and external enterprise systems. Connectors represent relevant integration relationships at a high level and should not be interpreted as complete or strictly directional data-flow specifications. In practice, individual workflows may involve synchronous API calls, asynchronous events, platform triggers, and bidirectional data exchange. These execution details are described in the following subsections.

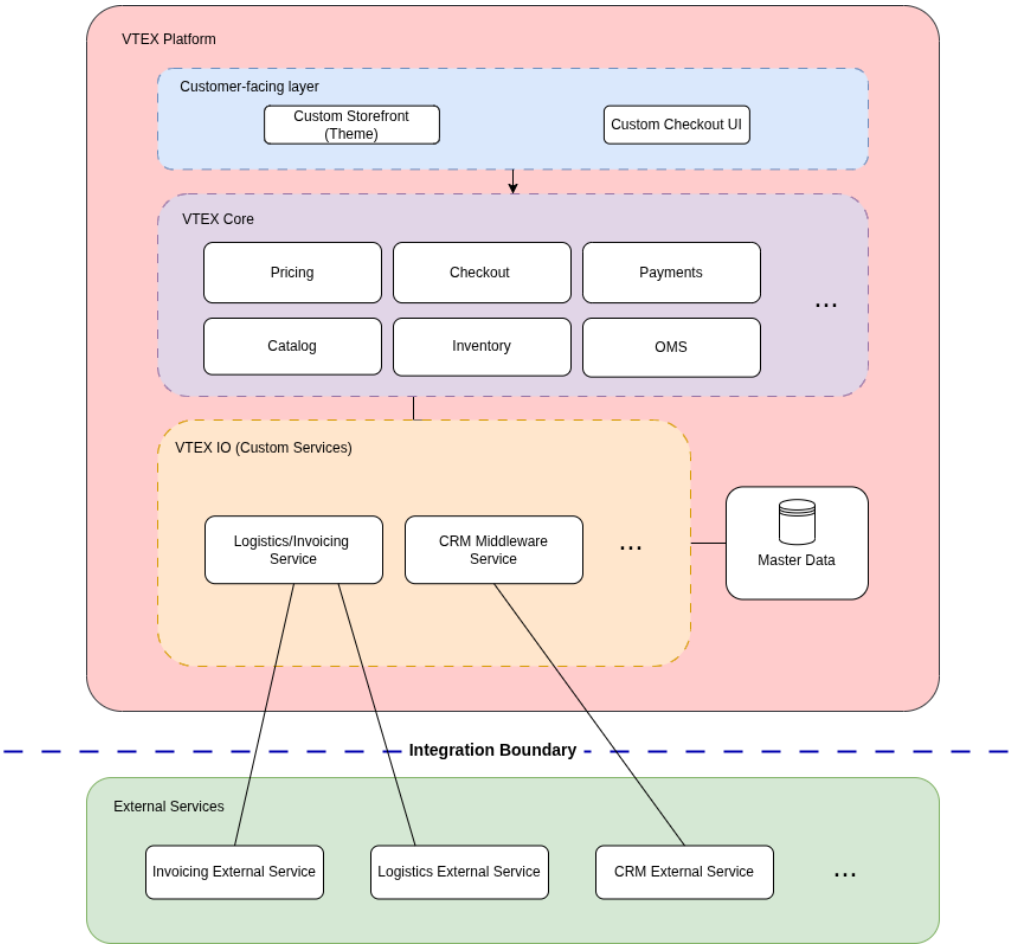



Figure 5.1: Simplified layered architecture of the VTEX-based solution, highlighting the main customer-facing components, VTEX Core services, VTEX IO custom services, Master Data, and external enterprise systems.

### 5.2.1 Custom Storefront

The storefront is the main entry point for users before starting the checkout process. It is built using VTEX IO frontend capabilities and follows a configuration-based approach, where page layouts are defined using JSON (JavaScript Object Notation) and rendered through React components.

Each page is composed of reusable blocks, which are dynamically assembled at runtime. This results in a flexible but complex DOM structure, often deeply nested and with automatically generated CSS (Cascading Style Sheet) class names. In addition, many components load asynchronously, depending on data fetched from VTEX APIs.

From a testing perspective, this makes element selection and UI validation inherently unstable if conventional approaches are used. Tests cannot rely on static CSS selectors, which are prone to breaking with any layout change, and must instead use more resilient strategies such as custom attributes when possible. They also need to handle asynchronous rendering explicitly to avoid flaky results caused by components not yet being mounted when assertions are evaluated.

### 5.2.2 Custom Checkout UI

Although VTEX provides a native checkout module, this project required a customized user experience. A custom frontend application was developed to extend and modify the default checkout behavior through additional scripts and UI changes.

This component interacts closely with VTEX checkout APIs and operates in a more controlled and synchronous manner compared to the storefront, which reduces some of the asynchronous rendering complexity encountered elsewhere.

Nevertheless, since checkout is the most critical step in the user journey — failures here directly prevent order creation — testing must achieve a higher degree of reliability than in other components. Every interaction and state transition must be validated with confidence, and test instability in this area carries a disproportionate cost.

### 5.2.3 CRM Middleware Service

The organization uses an external CRM system as the main source of truth for customer data. To keep this system synchronized with VTEX, a custom integration service was developed using Node.js.

This integration is based on triggers configured in VTEX Master Data. Whenever relevant changes occur — such as the creation or update of a customer record — Master Data sends a request to the service through a predefined endpoint. Upon receiving this request, the service processes the data and forwards it to the CRM system. This is a trigger-based model, where VTEX Master Data actively calls the service whenever specific events occur, rather than the service polling for changes or being called on demand.

Testing this integration introduces a qualitatively different challenge from the storefront or checkout: the behavior being validated is not directly triggered by a user action but occurs asynchronously in the background after one. This requires tests to wait for eventual consistency between systems and to verify state changes through indirect observation — for example, querying the CRM or Master Data after a delay — rather than asserting immediate outcomes.

### 5.2.4 Logistics/Invoicing Service

Order processing is handled through a custom service that integrates with external logistics and invoicing systems. This service centralizes multiple workflows, each with a different execution model.

The logistics integration is event-driven and tied to the VTEX order lifecycle. When an order reaches a specific state, VTEX broadcasts an event that is handled by the service, which then prepares the required data and sends it to a third-party logistics partner to initiate fulfillment.

The invoicing process is decoupled from order creation. The service exposes an API endpoint responsible for issuing invoices, which is periodically triggered by an external scheduler. This same endpoint can also be called manually, which is particularly useful for testing: rather

than waiting for the scheduler to fire, tests can trigger the invoicing process directly and reduce execution time.

The inventory synchronization process runs periodically and updates product stock levels based on data retrieved from the logistics partner. Like invoicing, it is also exposed through an API endpoint, allowing on-demand triggering during tests.

These workflows combine synchronous API calls with asynchronous external processes. The ability to trigger them directly through API endpoints is an important design affordance that the testing strategy exploits to make otherwise scheduler-dependent behavior testable in a controlled way.

### 5.2.5 Implications for Testing

Taken together, the system architecture makes testing complete flows substantially more complex than in a self-contained application. A typical user journey — from browsing products to order fulfillment and invoicing — spans multiple components, external systems, and mixed execution models.

Because of this, tests cannot rely solely on immediate results. In many cases it is necessary to wait for background processes to complete or for data to propagate between systems before assertions can be made. To manage this, the testing approach incorporates polling mechanisms and makes deliberate use of API endpoints that allow asynchronous processes to be triggered on demand, reducing dependency on external schedulers and shortening execution time.

While mocking is a standard technique for isolating components in unit testing, it is deliberately minimized in this project. The goal is to validate complete, realistic flows — including platform-specific behavior such as Master Data trigger execution and cross-system data propagation — which cannot be meaningfully tested against simulated dependencies. This trade-off accepts slower and less deterministic tests in exchange for higher confidence that the system behaves correctly under real operating conditions.

## 5.3 Implementation of the Proposed Strategy

Having described the VTEX platform and the system architecture, this section explains how the proposed testing strategy was implemented in the case study. The implementation details are presented as the concrete instantiation of the generic strategy introduced in Chapter 4.

The primary objective is to demonstrate how the proposed testing strategy was translated into a robust, maintainable codebase using Cypress, and how it was successfully integrated into the development lifecycle.

The section begins by detailing the approach used to select the most critical business scenarios for testing, acknowledging the execution costs inherent to cloud-dependent testing. It then explores the architectural organization of the test repository, highlighting the deliberate separation between UI interactions, API workflows, and shared helper utilities.

Subsequent sections delve into the specific technical solutions engineered to mitigate the SaaS constraints identified previously — particularly the mechanisms developed for handling eventual consistency and test data management. Finally, the section presents concrete code implementations of both API workflows and browser-based user journeys, concluding with the evaluation of the Contract Testing proof of concept and the configuration of the Continuous Integration (CI/CD) pipeline.

### 5.3.1 Environment Strategy

The shared-resource limitations of VTEX workspaces, described in Section 5.1.2, make it unsafe to run tests in the same account as production. Data created or modified during test execution could corrupt production records or interfere with real user activity.

To address this, since the very beginning of the project, a dual-account strategy has been implemented. One account is used exclusively for development and testing (DEV), while the other is reserved for production (PRD). Both accounts share the same configuration, minimizing the risk of environment-specific discrepancies that could cause tests to pass in development but fail in production. Within the development account, multiple workspaces can be created freely, allowing parallel development and experimentation without risk to production data.

This approach provides environment isolation at the account level, compensating for the lack of full isolation within workspaces. Moreover, the external systems involved in the tested workflows also provide separate testing environments with configurations aligned with production, ensuring that tests can interact with realistic dependencies without risking data integrity.

### 5.3.2 Test Design and Scenario Selection

Because end-to-end tests are expensive to execute and maintain in a cloud environment, attempting exhaustive test coverage across the entire platform is not feasible. Instead, a pragmatic, risk-based testing strategy was adopted. Test scenarios were prioritized based on their business criticality: flows that directly impact revenue generation (such as checkout and payments) and flows that sustain daily logistics (such as inventory synchronization and third-party fulfillment).

Furthermore, the strategy aimed to separate concerns based on the execution layer. The User Interface (UI) was reserved strictly for critical user-facing journeys, whereas complex business logic variations and edge cases were pushed to the API layer to maximize execution speed and reduce flakiness.

Table 5.1 details the critical business scenarios selected for automation, categorizing them by their execution layer and the specific logic variations they evaluate. As demonstrated, variations of a specific flow (such as shipping a fragile item versus a pickup order) were deliberately validated via API integration tests rather than UI workflows. This approach ensures robust validation of the generated partner artifacts (e.g., XML files) without incurring the performance and stability penalties of driving the browser through the entire purchase funnel multiple times.

| Business Capability | Layer | Evaluated Variations | Testing Objective |
|---|---|---|---|
| **Storefront Checkout Flow** | UI | Standard delivery; pickup point delivery | Validate the end-to-end user experience from adding a product to the cart through the successful rendering of the final payment step and order settlement, handling dynamic DOM rendering. |
| **Customer Product Registration** | UI | Coffee machine registration | Validate the authenticated user portal functionality, ensuring customers can successfully register and link hardware to their CRM profiles. |
| **Order Lifecycle and Fulfillment** | API | Full processing flow; order placement via Multibanco | Programmatically place orders and simulate external system responses to validate state transitions through handling, shipping, and eventual invoicing. |
| **Logistics Data Generation** | API | Fragile items for mainland delivery; fragile items for islands delivery; pickup point orders | Verify that the XML payloads generated for the Third-Party Logistics partner contain the correct flags, addresses, and structures for specific edge cases. |
| **Customer Identity and CRM Sync** | API | Account creation; account update | Validate the asynchronous trigger from Master Data to the external CRM, ensuring customer data is synchronized. |
| **Inventory Synchronization** | API | Inventory updates | Assert that invoking the scheduled stock-sync endpoint successfully updates product availability in the native platform catalog. |

Table 5.1: Selection of critical test scenarios, highlighting execution layers and logical branches.

While Table 5.1 outlines the high-level business capabilities selected for automation, the implementation of each capability may include multiple concrete test cases, assertions, input variations, and edge cases. In total, these critical flows were translated into a consolidated suite of 31 automated test cases: 5 browser-based end-to-end test cases and 26 API-driven test cases targeting backend logic and integrations.

However, this dissertation discusses the suite primarily at the level of business flows rather than individual assertions or low-level test cases. From this perspective, the 31 automated test cases cover 11 broader business flow variations (8 API-based and 3 UI-based E2E). This level of granularity better reflects the strategic intent of the suite, since the objective is not to maximize the number of test cases, but to provide targeted confidence in the most critical workflows of the system.

Rather than aiming for an arbitrary metric of code coverage, this volume represents a targeted defense against critical regressions, striking a balance between high business confidence, manageable CI/CD execution times and time required to implement the suite, considering the deadlines of this dissertation. Thus, the implementation covers the most relevant fows, having in mind the time and resource constraints of the project, while still providing a solid foundation for future expansion if needed.

### 5.3.3 Testing Architecture and Project Organization

Cypress was chosen as the primary testing framework for this implementation because it supports both API-level and browser-based end-to-end tests within the same tool. This was particularly useful in the context of the proposed strategy, since the test suite needed to combine backend-oriented workflow validation with selective UI journeys while sharing setup logic, fixtures, configuration, and helper utilities across both layers. The choice was also aligned with the project's TypeScript and Node.js ecosystem, reducing adoption effort and keeping the automation stack close to the existing development environment. Although alternative tools such as Playwright could also support similar goals, Cypress provided a mature ecosystem, strong documentation, and a pragmatic path to implement the required test suite within the available project constraints.

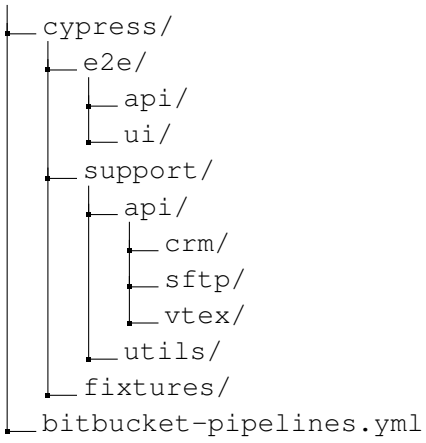



Figure 5.2: Directory structure enforcing the architectural separation of concerns.

To support the testing scenarios outlined in Section 5.3.2, the automation repository was structured to enforce a strict separation of concerns, in order to not interleave business assertions, platform-specific setup logic, and HTTP configurations within the same script. This way, the test specifications remain clean and focused on validating business rules, while the complexities of interacting with the VTEX platform are abstracted away in a shared helper layer. Moreover, the overall code gets organized in a way that promotes reusability and maintainability, as common logic is centralized rather than duplicated across test files.

To enforce this separation, the Cypress project was divided into distinct layers. Figure 5.2 illustrates the directory structure that physically enforces these boundaries.

As illustrated above, the test portfolio is not a homogeneous block, but rather a collection of specialized "test families" and support structures, each solving a different problem:

- **Browser-based E2E tests (`cypress/e2e/ui/`):** This family is used when the objective is to validate the actual user journey. It tests the system from the customer's perspective,

handling page rendering, client-side navigation, form interactions, and visual state transitions.

- **API / Integration tests (`cypress/e2e/api/`):** This is the dominant family in the suite, used whenever the system can be exercised more reliably directly through HTTP endpoints. By bypassing the browser, it validates backend flows, partner file generation, and order state transitions significantly faster and more deterministically.
- **Shared helper layer (`cypress/support/` and `cypress/fixtures/`):** This abstraction layer encapsulates platform-specific complexities. The API helpers are physically subdivided by target system (`vtex`, `crm`, and `sftp`) to maintain strict domain boundaries. Auxiliary tools, such as eventual consistency polling and XML parsers, are centralized in the `utils` directory. Furthermore, all static test data payloads are isolated in `fixtures` to keep the execution scripts clean.

By relying on the shared helper layer to handle VTEX-specific and third-party technicalities, the scripts in the execution layer remain declarative. Consequently, the test files read as high-level business scenarios rather than low-level implementation scripts, drastically improving their readability and maintainability. Finally, the execution of this architecture is orchestrated by the CI/CD configuration (`bitbucket-pipelines.yml`), the mechanics of which are detailed in Section 5.3.7.

### 5.3.4 Addressing Cloud Constraints: The Helper Layer

As an instantiation of the problem characterized in Chapter 3, the cloud nature of the VTEX platform and its surrounding ecosystem introduces two severe constraints for automated testing: the asynchronous eventual consistency of external integrations, and the lack of deterministic environment resets. The abstraction layer of the Cypress project was explicitly engineered to resolve these issues.

#### 5.3.4.1 Managing Eventual Consistency via Polling

In a microservices architecture, operations such as order processing and logistics dispatch are handled asynchronously through event buses. Consequently, immediate assertions in an end-to-end test will inevitably fail, as the data has not yet propagated across systems. A prime example in this project is the generation of the third-party logistics (3PL) delivery number, which is asynchronously written back to VTEX Master Data after an order is placed.

To prevent test flakiness while avoiding arbitrary and inefficient hardcoded pauses (`cy.wait()`), custom recursive polling mechanisms were implemented within the helper layer. Listing 5.1 demonstrates the implementation of the logistics polling function.

Listing 5.1: Recursive polling implementation to handle asynchronous 3PL data propagation.

```
import { VTEX_CONFIG } from './vtexConfig';

/**
 * Retrieves the delivery number for an order ID.
 * Retries recursively if the number is not available yet due to async processing.
 */
export const getRangelDeliveryNumber = (orderId, maxRetries = 15, delay = 3000) => {
  const attemptGet = (attempt = 0) => {
    return cy.request({
      method: 'GET',
      url: `${VTEX_CONFIG.BASE_URL}/api/dataentities/OI/search`,
      headers: VTEX_CONFIG.HEADERS,
      qs: { _fields: 'rangelDeliveryNumber,id', _where: `orderId=${orderId}` }
    }).then((response) => {
      expect(response.status).to.eq(200);

      // Check if data is missing or empty, triggering recursion if within limits
      const isMissing = !response.body || response.body.length === 0;
      const isRangelEmpty =
        !isMissing &&
        (!response.body[0].rangelDeliveryNumber ||
          response.body[0].rangelDeliveryNumber === '');

      if (isMissing || isRangelEmpty) {
        if (attempt < maxRetries) {
          cy.log(`Rangel number pending, retrying... (${attempt + 1}/${maxRetries})`);
          return cy.wait(delay).then(() => attemptGet(attempt + 1));
        } else {
          throw new Error(`Timeout: No delivery number found for order ${orderId}`);
        }
      }

      cy.log(`Rangel delivery number found: ${response.body[0].rangelDeliveryNumber}`);
      return cy.wrap(response.body[0].rangelDeliveryNumber);
    });
  };

  return attemptGet(0);
};
```

By wrapping Cypress's asynchronous `cy.request` in a recursive closure, the test intelligently yields to the platform's processing time. It continuously polls the Master Data entity until the expected payload is present, failing only if the maximum retry threshold is breached. This

ensures execution is as fast as the system allows, yet resilient to network or processing latency.

#### 5.3.4.2 State Management and Data Teardown

The second major constraint is the shared nature of cloud environments. Because tests run against a persistent database, data created in one test run can pollute subsequent executions. For instance, the UI test responsible for validating customer machine registration requires a clean slate; if a specific machine serial number is already registered to the test account from a previous run, the UI flow will fail.

Since local database resets are impossible, the testing strategy dictates that, whenever needed, the environment must be programmatically cleaned before the browser is even launched. This is achieved by combining API helpers inside setup hooks, as shown in Listing 5.2.

Listing 5.2: Idempotent test setup using CRM API helpers to prevent data pollution.

```
import { findMachineId, desassociateMachine } from '../../support/api/crm/
    crmHelpers';
import { DELTA_ACCOUNT } from '../../support/api/vtex/vtexConfig';

describe('Customer Asset Registration Flow', () => {
  // Idempotent environment setup
  beforeEach(() => {
    cy.on('uncaught:exception', () => false);

    // Communicate directly with the external CRM to reset test state via API
    findMachineId(DELTA_ACCOUNT.accountNumber, MACHINE_SERIAL_NUMBER)
      .then((machineId) => {
        if (!machineId) return; // State is already clean
        return desassociateMachine(machineId); // Teardown previous state
      });

    // Proceed to visit the storefront UI...
  });

  it('should allow a user to register a new coffee machine', () => {
    // UI interactions begin here in a clean state
  });
});
```

This pattern enforces test idempotency. By offloading the state teardown to the API layer before the UI interactions begin, the test suite guarantees some determinism without relying on manual database interventions. This hybrid approach — using backend APIs to orchestrate the conditions for frontend validations — is central to the reliability of the entire test repository.

### 5.3.5 Implementation of Core Test Families

With the architectural foundation established and the helper layer mitigating platform-level latency, the actual test specifications can focus entirely on validating business logic. As categorized in Section 5.3.3, these specifications are divided into API-driven workflows and browser-based E2E UI journeys.

#### 5.3.5.1 API and Integration Workflows

The most complex workflows in this VTEX e-commerce project — such as order fulfillment, XML generation for logistics, and inventory synchronization — occur downstream, after an order is placed. Validating these downstream processes requires an existing, valid order in the system.

If the test suite relied on the storefront UI to generate these prerequisite orders, execution times would skyrocket, and the backend tests would become vulnerable to unrelated frontend rendering flakiness. To prevent this, a programmatic, headless checkout flow was engineered. In VTEX, the checkout process happens with multiple API calls that create and modify the order form: cart creation, item addition, profile attachment, shipping data submission, and payment processing. By orchestrating these calls directly through the API helpers, this is the fastest way to generate a valid order in the system, in less than 10 seconds (depending on latency and network conditions).

Listing 5.3 illustrates the `createCompleteOrder` helper, which chains multiple VTEX API endpoints to simulate a complete purchase without ever launching a browser.

Listing 5.3: Programmatic execution of the VTEX checkout pipeline via API, bypassing the UI.

```
export const createCompleteOrder = (
  items,
  deliveryAddress,
  giftCardCode,
  paymentSystem = 'giftCard',
  deliverAtPickupPoint = false
) => {
  let orderFormId, totalValue, transactionResult;

  return createOrderForm()
    .then((id) => {
      orderFormId = id;
      return addItems(orderFormId, items);
    })
    .then(() => attachClientProfile(orderFormId))
    .then(() => attachShippingData(orderFormId, deliveryAddress,
        deliverAtPickupPoint))
    .then(() => attachPaymentData(orderFormId, paymentSystem, giftCardCode))
    .then((total) => {
      totalValue = total;
      return createTransaction(orderFormId, totalValue);
```

```
21      })
22      .then((txResult) => {
23        transactionResult = txResult;
24        return sendPaymentInfo(transactionResult, paymentSystem);
25      })
26      .then(() => processGatewayCallback(transactionResult.orderGroup))
27      .then(() => {
28        if (paymentSystem === 'multibanco') {
29          return confirmOrderSuccess(transactionResult.orderGroup);
30        }
31      })
32      .then(() => {
33        cy.log('Order completed successfully via API!');
34        return cy.wrap({
35          orderFormId,
36          orderId: transactionResult.orderGroup + "-01",
37          items: transactionResult.items,
38          shippingData: transactionResult.shippingData
39        });
40      });
41  };
```

By utilizing JavaScript promises to sequentially orchestrate the cart, profile, shipping, and payment gateways, this helper provisions a fully valid order in seconds. With the prerequisite state instantly generated, the integration tests can strictly focus on asserting downstream behaviors rather than wasting compute time on storefront navigation.

The most critical example of this in the project is the Full Order Lifecycle and Logistics synchronization, which involve many asynchronous steps and interactions between multiple microservices (both VTEX core services, custom VTEX IO services/apps, and external services that we do not control). Using the headless order as a starting point, this test acts as an orchestrator across multiple systems, following a strict pipeline. Let's consider the full flow of an order, from placement to invoicing:

1. **State Generation:** A headless order is generated using the `createCompleteOrder` helper.

2. **Eventual Consistency:** The test polls the VTEX platform until the order status transitions to `handling`.

3. **Outbound File Validation:** The test connects to the SFTP (Secure File Transfer Protocol) server, retrieves the generated outbound XML file, and parses its structure to validate headers, shipment information, and the item list.

4. **Partner Writeback Simulation:** To test the inward flow, the test uploads a mock inbound XML file to the SFTP and invokes the platform's scheduler.

5. **Final Reconciliation:** The suite polls the system until the order is marked as `invoiced`, validating the final invoice payload in Master Data.

This specification serves as the primary example of a strict integration test in the repository. It asserts the collaboration of multiple systems (VTEX core, Master Data, SFTP, and 3PL logic) without ever relying on browser rendering. Furthermore, by encapsulating middle-tier logic within the shared helper layer, the test specification remains readable and strictly focused on business rules.

#### 5.3.5.2 UI-based E2E Workflows

While the API layer handles deep business logic efficiently, the storefront and checkout must be validated from the user's perspective. However, as established in Section 5.2, testing the storefront introduces significant client-side challenges. The architecture heavily relies on React for asynchronous component rendering and lazy loading. Moreover, considering that we can only run tests against a "live" environment (even though it is reserved for testing purposes), even minor network fluctuations or background processes can cause components to render slower than usual, leading to transient errors in the test suite.

If a test runner executes commands faster than the browser can mount these components, tests will inevitably fail with "element not detached" or "not clickable" errors. To mitigate this, the UI automation strategy shifted away from implicit framework waits toward explicit state assertions and more deterministic DOM manipulation.

Listing 5.4 illustrates a segment of the end-to-end checkout journey, specifically highlighting the strategies used to overcome frontend instability during product selection and pickup-point choice.

Listing 5.4: Handling lazy-loaded components, dynamic pricing, and complex modal interactions in the UI checkout flow.

```
1  it('should allow anonymous user to complete a pickup purchase', () => {
2    cy.goToHomepage();
3
4    // 1. Handling Lazy Loading & DOM Stability
5    cy.get('a.vtex-product-summary-2-x-clearLink', { timeout: 50000 })
6      .first().should('be.visible').click();
7
8    cy.document().its('readyState').should('eq', 'complete');
9
10   // Force component mounting to ensure "Add to Cart" is fully interactable
11   cy.scrollTo('bottom');
12   cy.get('form.thedeltahousedev-store-theme-0-x-newsletterWithConsent', { timeout:
         50000 })
13     .should('be.visible');
14
```

```
  cy.get('span.vtex-add-to-cart-button-0-x-buttonText')
    .contains('Adicionar ao carrinho').click();

  // 2. Data-Agnostic Assertions (Regex)
  // Prevents test failures if product prices change in the catalog
  cy.get('div.vtex-checkout-summary-0-x-price', { timeout: 50000 })
    .should('be.visible')
    .invoke('text')
    .should('match', /€\s*\d+(.\d{2})?/);

  cy.get('button#proceed-to-checkout').should('be.visible').click();

  // ... (Navigates to shipping step) ...

  // 3. Complex UI Component Handling (External API Dropdowns)
  cy.get('button#shipping-option-pickup-in-point').should('be.visible').click();
  cy.get('button#find-pickups-manualy-button').click();

  cy.get('form.vtex-pickup-points-modal-3-x-modalSearch input')
    .type(testData.address.postalCode);

  // Wait for the third-party (Google Places) dropdown to populate
  cy.get('div.pac-container.pac-logo', { timeout: 50000 })
    .should('have.length.gte', 1);

  cy.get('div.pac-container.pac-logo div.pac-item')
    .first().should('be.visible').click();

  // Assert the VTEX pickup list updates before selecting a location
  cy.get('div.vtex-pickup-points-modal-3-x-pointsList')
    .find('div.vtex-pickup-points-modal-3-x-pointsItem')
    .should('have.length.gt', 1);
});
```

As demonstrated in the code, the suite employs several defensive engineering practices:

- **Render Forcing:** The `cy.scrollTo('bottom')` command is deliberately utilized to trigger lazy-loaded components (such as the footer). By asserting the visibility of this final component, the test guarantees that the React execution context has settled before attempting critical interactions like adding an item to the cart.
- **Data-Agnostic Assertions:** UI tests are highly vulnerable to background data changes. By asserting prices and order numbers using Regular Expressions (e.g., `/\d+(\.\d2)?\s*€/`) rather than hardcoded string matching, the test becomes resilient to catalog price updates or promotions.
- **Eventual Consistency in the DOM:** When interacting with the pickup-point modal, the UI relies on an external mapping API. The test script explicitly waits for the generated

`pac-container` class to yield results, chains a length assertion to guarantee the array is populated, and only then simulates the user click.

Moreover, adding a timeout inside the `cy.get()` command ensures that the test runner patiently waits for the component to render, rather than failing immediately if it is not present. This is crucial in a cloud environment where rendering times can be unpredictable and Cypress defaults may not be sufficient (due to the specific constraints of this context). The adjustment of these timeouts was based on empirical observations of the platform's performance, striking a balance between test execution time and stability. Thus, it was an iterative and continuous process of tuning these parameters to find the optimal configuration that minimizes flakiness without unnecessarily prolonging test runs.

Through these deterministic assertions, the UI testing suite successfully models complex user journeys while mitigating the inherent fragility of cloud-hosted, dynamically rendered applications.

### 5.3.6 Contract Testing Evaluation

Contract testing was evaluated as a potential complement to the integration and system-level tests implemented in this dissertation. In principle, CDCT allows communicating services to be tested independently against a shared contract, reducing the need for both systems to be available simultaneously during some validation activities. This makes it attractive in distributed systems, where integration failures may result from mismatches between consumer expectations and provider behavior.

However, the applicability of CDCT in the project architecture is not uniform across all integration boundaries. For external third-party integrations, such as the CRM and third-party logistics provider, contract testing would require active participation from both the consumer and the provider. In practice, enforcing shared contracts with external enterprise vendors is difficult because the development team has no administrative control over their testing pipelines, release processes, or API evolution.

For internal integrations between custom VTEX IO services, the situation is different. Both sides of the communication are maintained within the same development context, which reduces the organizational benefit normally associated with CDCT. A major motivation for CDCT is to support independent evolution across autonomous teams; however, in this project, coordination between the relevant services can still be handled through direct internal communication. If the project were to scale to multiple independent teams owning separate services, CDCT would become a more justifiable investment.

Despite these limitations, an exploratory implementation of CDCT was developed using the Pact framework. The objective was not to replace the existing integration tests, but to evaluate whether CDCT could provide additional value by isolating one service boundary and exposing the practical challenges of applying contract testing in a proprietary SaaS context. This evaluation also provides a concrete implementation example, helping to assess aspects that are less visible

in purely theoretical discussions, such as Pact Broker usage, provider verification, and the setup required to reproduce provider states.

#### 5.3.6.1 The Intended Architecture

The theoretical appeal of CDCT is that it allows two communicating services to be tested independently. Instead of deploying both services to an integrated environment, they agree on a strictly defined JSON contract.

To evaluate this approach, the implementation targeted an internal integration between the VTEX Storefront Theme, acting as the Consumer, and the CRM middleware service, acting as the Provider. Specifically, the evaluation focused on the endpoint responsible for retrieving the machines registered by a customer. Figure 5.3 presents a sequence diagram of the intended CDCT workflow, showing how the consumer-side contract would be generated, published to the Pact Broker, and later retrieved and verified against the provider service.

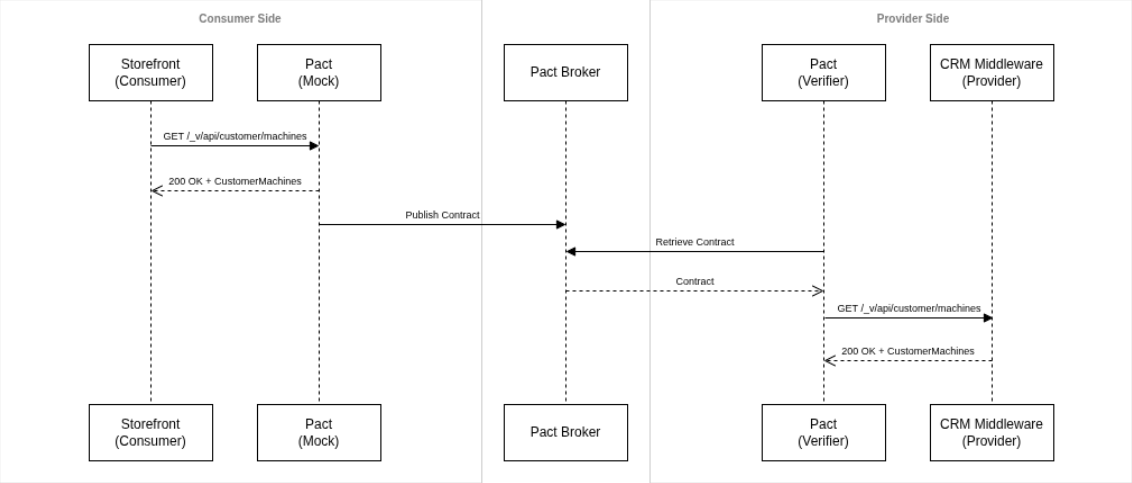



Figure 5.3: Sequence diagram of the intended CDCT workflow, including consumer-side contract generation, publication to the Pact Broker, and provider-side verification.

To implement this, there are three mains parts involved: the Consumer, the Pact Broker, and the Provider. The following sections detail the evaluation of each component and its compatibility with the project's cloud constraints.

#### 5.3.6.2 The Consumer Implementation

The first step of the pipeline is defining the expectations from the consumer's perspective. In the Theme repository, a Pact test was created to specify the exact data structure expected from the CRM middleware service.

Listing 5.5 demonstrates this implementation using the `PactV3` API. The test utilizes matchers (`MatchersV3.eachLike`), ensuring the contract enforces data types rather than exact, hard-coded values.

Listing 5.5: Pact Consumer test defining the expected GET interaction. The focus is strictly on the contract definition rather than response assertions.

```
const test = require('node:test');
const axios = require('axios');
const { PactV3, MatchersV3 } = require('@pact-foundation/pact');
const { pactConfig } = require('../setup/pactConfig');

const provider = new PactV3(pactConfig);

test('GET /customer/machines returns CustomerMachines list', async () => {
  // 1. Define the Contract Expectations
  provider
    .given('customer has associated machines')
    .uponReceiving('a request to list customer machines')
    .withRequest({
      method: 'GET',
      path: '/_v/api/customer/machines',
    })
    .willRespondWith({
      status: 200,
      headers: { 'Content-Type': 'application/json; charset=utf-8' },
      body: {
        CustomerMachines: MatchersV3.eachLike({
          machineId: MatchersV3.string('machine-123'),
          serialNumber: MatchersV3.string('SN-123456'),
        }),
      },
    });

  // 2. Execute the HTTP client to generate the contract
  await provider.executeTest(async (mockServer) => {
    await axios.get(`${mockServer.url}/_v/api/customer/machines`);
  });
});
```

Executing this test successfully generates a Pact contract file. Listing 5.6 shows an excerpt of the generated contract, including the expected request, provider state, response schema, and matching rules. These matching rules are particularly relevant because they demonstrate that the contract does not require the exact hardcoded values used in the example response. Instead, it verifies the expected structure and data types of the response returned by the provider.

Listing 5.6: Excerpt of the generated Pact contract for the customer machines endpoint.

```
{
  "consumer": {
    "name": "storefront-theme"
```

```
  },
  "provider": {
    "name": "crm-middleware"
  },
  "interactions": [
    {
      "description": "a request to list customer machines",
      "providerStates": [
        {
          "name": "customer has associated machines"
        }
      ],
      "request": {
        "method": "GET",
        "path": "/_v/api/customer/machines"
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": {
          "CustomerMachines": [
            {
              "machineId": "machine-123",
              "serialNumber": "SN-123456"
            }
          ]
        },
        "matchingRules": {
          "body": {
            "$.CustomerMachines": {
              "combine": "AND",
              "matchers": [
                {
                  "match": "type",
                  "min": 1
                }
              ]
            },
            "$.CustomerMachines[].machineId": {
              "combine": "AND",
              "matchers": [
                {
                  "match": "type"
                }
              ]
            },
            "$.CustomerMachines[].serialNumber": {
```

```
                "combine": "AND",
                "matchers": [
                  {
                    "match": "type"
                  }
                ]
              }
            }
          }
        }
      }
    ]
}
```

The generated contract confirmed that the consumer-side implementation was able to capture the expected interaction with the CRM middleware without depending on the actual provider service during test execution.

#### 5.3.6.3 The Pact Broker

In a continuous integration environment, the generated contract is typically published to a Pact Broker. The Pact Broker acts as a centralized repository for sharing consumer contracts and provider verification results.

However, introducing a Pact Broker would require additional infrastructure and operational configuration. Although PactFlow [19] provides a managed cloud-hosted alternative to self-hosting the open-source Pact Broker, relying on this service would introduce an additional external dependency and, depending on the required usage tier, potential additional costs.

Since the primary goal of this work was to evaluate the compatibility of the Pact framework with the VTEX platform and its execution constraints, the setup and maintenance of a dedicated Pact Broker infrastructure was considered outside the scope of the practical evaluation.

#### 5.3.6.4 Provider Verification Constraints

While the consumer-side implementation was relatively straightforward, the practical evaluation encountered limitations during the provider verification phase. These limitations are mainly related to how contract testing frameworks rely on controlled provider states, in contrast with broader integration tests that exercise the system through externally visible business flows.

In many conventional CDCT setups, the provider service is executed locally or in a controlled test environment during verification. In such environments, the verifier can prepare the required provider states before replaying the interactions described in the contract. For example, a state such as “a customer has associated machines” can be established through setup code, database fixtures, or controlled test data before the Pact verifier sends the corresponding HTTP request.

However, this approach is difficult to reproduce in the VTEX IO architecture. The CRM middleware is a serverless application that must be deployed to a VTEX cloud workspace to function

in an environment equivalent to the real execution context. Consequently, a local test runner cannot directly control or intercept the internal state of the deployed service or of the cloud-hosted data sources on which it depends.

A possible workaround would be to expose a dedicated provider state setup endpoint, such as `/_pact/setup`, in the deployed application. Before each provider verification interaction, the verifier could call this endpoint to instruct the application to create or configure the required test data in the cloud environment.

Although this approach could be technically feasible within a development workspace, it introduces a significant maintenance burden and an architectural compromise. Unlike the API tests described in Section 5.3.5, which prepare test data through natural business flows and public endpoints, this workaround would require developers to implement and maintain artificial setup controllers inside the application repository. These controllers would serve no business purpose other than preparing internal state for the contract testing framework.

Furthermore, one of the practical benefits commonly associated with CDCT, namely fast and isolated provider verification, would be reduced in this context. Since the provider verification would still depend on a fully deployed VTEX cloud workspace and additional API calls to prepare provider states, the expected execution-time advantage over the already implemented API-level integration tests could be significantly weakened. For these reasons, full provider-side Pact verification was considered disproportionately costly in relation to the goals and constraints of this work.

#### 5.3.6.5 Conclusion of the Evaluation

The exploratory implementation showed that Pact can be used to define and generate consumer-side API contracts in the evaluated VTEX context but in a very limited way. The consumer-side test was straightforward to implement and successfully captured the expected interaction between the storefront and the CRM middleware endpoint.

However, extending the approach to full provider-side verification revealed important practical limitations. In particular, the isolated provider state management required by CDCT is difficult to reproduce in a VTEX IO environment, where the provider service must run in a deployed cloud workspace and depends on platform-managed data sources and external services. Supporting this verification process would likely require dedicated provider state setup mechanisms, such as artificial test-only endpoints for injecting or configuring data before each verification.

Although technically feasible, this would introduce additional maintenance effort and architectural compromise. Unlike the API-level integration tests implemented in this work, which prepare data through externally visible business flows and public endpoints, the Pact provider verification setup would require test-specific mechanisms inside the application codebase. In this context, the expected benefits of isolated contract verification were considered insufficient to justify the additional complexity.

As a result, the full Pact pipeline, including automated provider verification through a Pact Broker, was not implemented. Instead, the evaluation was limited to assessing consumer-side

contract generation and identifying the constraints that make full CDCT adoption challenging in a serverless, proprietary SaaS environment such as VTEX.

### 5.3.7 Continuous Integration and Pipeline Automation

An automated test suite only delivers continuous value if it is integrated into the development lifecycle. In this project, the execution is fully automated using Bitbucket Pipelines. The pipeline strategy is designed around the architectural constraints of the VTEX platform — specifically the need to test against cloud-deployed workspaces — and the decision to maintain a centralized testing repository.

#### 5.3.7.1 Testing Against Cloud Workspaces

Unlike traditional web development, VTEX storefronts and background services cannot be fully built and tested on a local machine. Developers must deploy their branch changes to a cloud-based sandbox known as a VTEX Workspace.

To ensure the automated tests validate the correct code, the application pipelines automatically extract the developer's branch name, sanitize it, and pass it as an environment variable (`VTEX_WORKSPACE`). The Cypress tests then use this variable to target the exact cloud environment where the new features are deployed, rather than running against the production environment.

#### 5.3.7.2 Cross-Repository Trigger Flow

Because the end-to-end tests are housed in a dedicated repository (as detailed in Section 5.3.3), a standard single-repository pipeline is insufficient. Instead, a cross-repository trigger mechanism is used.

When a developer creates a Pull Request in an application repository (e.g., the storefront theme), that repository's pipeline runs its own unit tests and then triggers the central testing repository via the `atlassian/trigger-pipeline` pipe. Crucially, the application pipeline is configured to wait (`WAIT: 'true'`) for the Cypress tests to finish. If the central tests fail, the developer's Pull Request is blocked from being merged into the `main` branch.

#### 5.3.7.3 Selective Execution and Security

Running the entire test suite on every commit is inefficient. To optimize cloud compute time and provide faster feedback, the testing repository exposes separate custom pipelines for different test families. For example, if a change is made in a backend service repository, it triggers only the API tests; if a change is made in the storefront theme, it triggers the UI tests.

Listing 5.7 illustrates a simplified version of the testing repository's YAML configuration, showing how different pipelines are exposed and how environment variables are handled.

Listing 5.7: Bitbucket Pipelines configuration in the central testing repository, illustrating selective test execution and secure variable injection.

```
pipelines:
  custom:
    run-api-tests:
      - variables:
          - name: VTEX_WORKSPACE
      - step:
          name: Run Cypress API Tests
          image: cypress/base:latest
          caches:
            - node
            - cypress
          script:
            - npm install
            - npx cypress run
                --spec "cypress/e2e/api/**/*.js"
                --env vtexWorkspace=$VTEX_WORKSPACE
                --env VTEX_APP_KEY_DEV=$VTEX_APP_KEY_DEV
                --env VTEX_APP_TOKEN_DEV=$VTEX_APP_TOKEN_DEV

    run-ui-tests:
      - variables:
          - name: VTEX_WORKSPACE
      - step:
          name: Run Cypress UI Tests
          image: cypress/base:latest
          caches:
            - node
            - cypress
          script:
            - npm install
            - npx cypress run
                --spec "cypress/e2e/ui/**/*.js"
                --env vtexWorkspace=$VTEX_WORKSPACE
          artifacts:
            - cypress/logs/**
            - cypress/screenshots/**
            - cypress/videos/**
```

As seen in the configuration, sensitive data such as VTEX API keys (`VTEX_APP_KEY_DEV`) are not hardcoded. They are injected securely at runtime using Bitbucket Secrets. To bridge the gap between the CI/CD environment and the test runner, these variables are explicitly mapped in the `cypress.config.js` file using Node's `process.env` object. By mapping secrets like `SFTP_API_KEY`, `CRM_API_KEY`, and login credentials directly into the Cypress environment object, the test scripts can access sensitive data securely and consistently, regardless of whether

they are executed locally by a developer or autonomously in the cloud pipeline. Furthermore, if a UI test fails, the pipeline automatically preserves the generated screenshots and video recordings as artifacts, ensuring the development team has the necessary visual context to debug storefront failures efficiently.

## 5.4 Summary

This chapter presented the VTEX case study and detailed the practical implementation of the proposed testing strategy. It first described the platform context, the system architecture, and the main custom components and integrations that define the system under test. It then showed how the testing suite was implemented as the concrete application of the generic strategy introduced in Chapter 4.

By adopting a risk-based approach, the test suite prioritizes high-value business flows while maintaining strict boundaries between UI and API execution layers. To overcome the challenges of the VTEX cloud ecosystem, custom abstraction layers were developed to handle asynchronous eventual consistency through recursive polling and to ensure deterministic test states via API-driven data teardowns. These foundations enabled the creation of highly efficient, headless integration tests for complex backend logistics, alongside resilient, browser-based UI journeys that gracefully handle dynamic frontend rendering. It is important to mention that sporadic use of AI tools was employed for code assistance, essentially for generating boilerplate code and refactoring. Nonetheless, everything was carefully reviewed and validated to ensure correctness and rigor.

Furthermore, an exploratory evaluation of Consumer-Driven Contract Testing concluded that the necessary isolated state management is architecturally misaligned with serverless cloud environments, reinforcing the value of the chosen API tests, considering the limitations of the project. Finally, the integration of these test families into Bitbucket Pipelines established a secure, cross-repository automation flow capable of dynamically targeting developer workspaces.

With the testing architecture fully realized and integrated into the continuous integration lifecycle, the following chapter will evaluate the tangible results of this solution, analyzing its impact on pipeline execution efficiency and reliability.

# Chapter 6

# Results and Discussion

This chapter presents and analyzes the data collected during the execution of the test suite detailed in the previous chapter. The primary objective is to evaluate the viability, efficiency, and reliability of the proposed hybrid test architecture within a distributed cloud environment, where multiple microservices interact to support complex business processes.

To mitigate the impact of momentary network anomalies and obtain stable estimates of central tendency and dispersion, the automated test suite was executed 20 independent times in a development environment. The evaluation focuses on 11 critical business scenarios, strategically divided between the API and the UI layers. It is fundamental to clarify that these 11 scenarios represent macroscopic business flows rather than atomic test cases. Within the Cypress framework, a single business scenario (mapped to a dedicated specification file) encapsulates multiple distinct test blocks (`it` statements) and sequential validation steps. This approach ensures that the collected metrics reflect the true complexity of end-to-end transactions rather than isolated system assertions.

Before presenting the data, it is important to explicitly define the primary metrics used to evaluate the test suite throughout this chapter:

- **Execution Time and Standard Deviation ($\sigma$):** The average duration (in seconds) required to complete either the entire CI pipeline or a specific business scenario. The standard deviation is included to quantify temporal volatility and evaluate the determinism of the tests.
- **Success Rate (%):** The percentage of independent runs that completed successfully without terminal errors. This is measured both at the macroscopic pipeline level and at the individual scenario level.
- **Framework Interventions (%):** The percentage of executions where Cypress's native retry mechanism was triggered due to some step failure. This metric serves as a practical, quantitative indicator of test fragility and environmental instability.

The analysis of the extracted data is structured into three main sections. Section 6.1 introduces the global execution metrics, providing a comprehensive overview of execution times and the

pipeline's architectural resilience. Section 6.2 provides a comparative analysis of the temporal costs associated with UI E2E testing versus API testing, using a common transactional objective as a baseline. Finally, Section 6.3 discusses the impact of asynchronous cloud architectures on test determinism and execution stability.

More granular data regarding the execution of the tests, from which the metrics presented in this chapter were derived, is available in Appendix A.

## 6.1 Overall Execution Metrics

To evaluate the viability of the proposed architecture within a real-world software development lifecycle, it is necessary to first analyze the test suite from a macroscopic Continuous Integration (CI) perspective. Table 6.1 summarizes the overall performance of the complete end-to-end pipeline (API and UI stages combined) across the 20 independent executions.

| **Pipeline Metric** | **Value** |
|---|---|
| Total Executions (Runs) | 20 |
| Average Total Execution Time | ∼13.6 minutes (820s) |
| Standard Deviation of Execution Time ($\sigma$) | ∼1.7 minutes (102s) |
| Overall Pipeline Success Rate | 90% (18/20 runs) |
| Pipeline Failures | 10% (2/20 runs) |

Table 6.1: Global CI/CD pipeline performance metrics across 20 executions.

This macroscopic view establishes a baseline level of reliability while simultaneously highlighting several architectural trade-offs. The overall success rate of 90% suggests that the pipeline is generally resilient under the evaluated conditions. Nevertheless, the 10% failure rate (comprising Run 18 in the API layer and Run 6 in the UI layer, see Appendix A) illustrates the inherent challenges associated with validating distributed microservices architectures, particularly in cloud-constrained environments. Analysis of the execution logs indicates that both failures were associated with transient timeout conditions in the underlying infrastructure rather than deterministic defects in the test implementation. This observation highlights the practical limitations of eventual consistency models in highly distributed systems, where asynchronous propagation delays may occasionally exceed the tolerance thresholds configured within automated validation workflows.

Furthermore, the average total execution time of approximately 13.6 minutes remains manageable for the current scope. However, it also introduces important considerations regarding future scalability. While a feedback cycle of approximately 13 minutes is generally acceptable within a standard agile development workflow, a substantial increase in the number of scenarios would likely increase pipeline execution times and reduce feedback efficiency unless additional optimization strategies, such as test parallelization, selective execution, or test suite partitioning,

are adopted. Consequently, future test additions should be carefully prioritized according to business criticality and risk exposure, balancing coverage requirements against execution cost and maintenance overhead.

It is imperative to contextualize these results within the broader landscape of distributed software systems. Microservices and distributed architectures inherently present significant testability challenges, with end-to-end (E2E) tests widely acknowledged in the literature as notoriously complex to maintain and execute reliably. This complexity is further exacerbated in this study by the reliance on a fully cloud-based, proprietary ecosystem. Operating with restricted access to the underlying infrastructure means that the automated suite is inevitably exposed to uncontrollable external variables, such as network latency, resource contention, and transient service degradation. Consequently, while the 90% overall success rate is a positive indicator of the pipeline's resilience, it must be evaluated against these adverse operational conditions. Achieving this level of determinism in a constrained environment is an encouraging outcome; nevertheless, it simultaneously serves as a pragmatic reminder of the unavoidable limitations and inherent unpredictability associated with testing distributed cloud architectures.

To better understand the factors contributing to the observed execution times, stability metrics, and environmental unpredictability, Table 6.2 provides a granular breakdown of the 11 business scenarios. Crucially, it introduces the Standard Deviation ($\sigma$) of the execution times, providing a direct quantitative measure of test determinism.

| **Business Scenario** | **Layer** | **Avg. Time $\pm$ $\sigma$ (s)** | **Success Rate (%)** | **Framework Interventions (%)** |
|---|---|---|---|---|
| Full Flow Order | API | $110.0 \pm 58.8$ | 95.0 | 5.0 |
| Order Multibanco | API | $13.1 \pm 2.5$ | 100.0 | 0.0 |
| Stock Update | API | $145.6 \pm 22.5$ | 100.0 | 15.0 |
| XML Mainland | API | $85.4 \pm 14.4$ | 100.0 | 10.0 |
| XML Islands | API | $82.2 \pm 15.3$ | 100.0 | 5.0 |
| XML Pickup | API | $28.2 \pm 2.4$ | 100.0 | 0.0 |
| Customer Creation | API | $8.2 \pm 3.7$ | 100.0 | 0.0 |
| Customer Update | API | $11.2 \pm 5.5$ | 100.0 | 0.0 |
| Checkout Standard | UI | $109.9 \pm 56.8$ | 95.0 | 35.0 |
| Checkout Pickup | UI | $84.7 \pm 42.9$ | 100.0 | 20.0 |
| Machine Registration | UI | $141.5 \pm 25.7$ | 100.0 | 10.0 |

Table 6.2: Detailed execution metrics per business scenario, including the Standard Deviation ($\sigma$) of execution times.

A closer examination of the individual scenarios reveals that execution time, stability, and standard deviation ($\sigma$) are strongly influenced by the architectural complexity of each transaction. Within the API layer, operations which involve less cross-service interactions such as *Order Multibanco* and *XML Pickup* exhibit low standard deviations ($\pm 2.5s$ and $\pm 2.4s$, respectively), suggesting that when relying on direct HTTP responses, the automated framework behaves more

predictably. Conversely, complex orchestrations like *Full Flow Order* ($\pm 58.8s$) and *Stock Update* ($\pm 22.5s$) present considerably higher deviations, consistent with the expected behavior of active polling mechanisms waiting for eventual consistency to resolve across multiple distributed services (VTEX Core, Custom Services, External Services, etc).

Furthermore, the results indicate a higher sensitivity within the UI layer. Scenarios like *Checkout Standard* not only required framework interventions (retries) in 35% of executions to achieve a successful outcome but also presented a high standard deviation of $\pm 56.8s$. This deviation is consistent with the unpredictable nature of browser testing in a cloud environment, where test duration is subject to asynchronous rendering behavior, external service response variability, DOM hydration, and third-party script loading. This observation reinforces the notion that UI-based end-to-end tests tend to be more fragile and less deterministic than API-level validations.

These findings are consistent with the architectural principle adopted throughout this work: whenever feasible, validation should be performed at lower testing layers, where execution is not only faster but generally more deterministic. UI testing remains indispensable for validating complete user journeys from an end-user perspective; however, relying extensively on UI tests for large-scale functional validation introduces significant maintenance overhead, higher deviation in execution times, and an increased susceptibility to flakiness.

## 6.2 API vs. UI E2E Execution Comparison

While the global metrics presented in Section 6.1 establish a broad correlation between the testing layer and execution stability, directly comparing scenarios with different business objectives can introduce analytical bias. To more rigorously quantify the overhead introduced by the User Interface (UI), it is necessary to isolate the testing layer as the primary independent variable.

This section focuses on a single, highly complex transactional objective common to both layers: the complete creation of a customer order. In the API suite, this objective is fulfilled by the *Order Multibanco* scenario, which bypasses the frontend and injects the necessary payloads directly into the integration endpoints. In the UI suite, the identical business objective is achieved through the *Checkout Standard* scenario, which simulates a real user navigating the browser, rendering the DOM (Document Object Model), and interacting with graphic elements.

Figure 6.1 visually illustrates the contrast in execution time and standard deviation between these two approaches.

The observed data reveals a substantial difference across all metrics. Achieving the "order created" state via direct API communication required an average of 13.1 seconds, with a relatively low standard deviation of $\pm 2.5s$. Furthermore, as detailed in Table 6.2, this API scenario required no framework interventions across any of the 20 executions, indicating consistent and stable behavior.

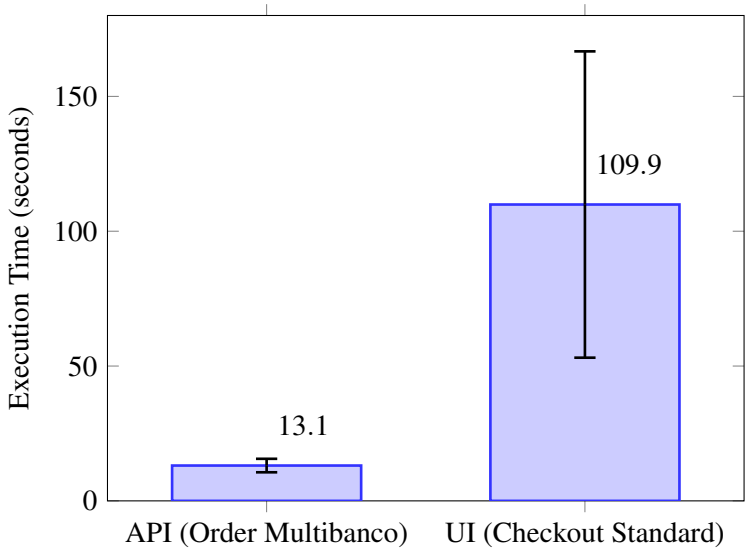



Figure 6.1: Performance and stability contrast for the "Order Creation" business flow. Error bars represent the standard deviation ($\sigma$).

Conversely, fulfilling the same business requirement through the UI required an average of 109.9 seconds — approximately 8.4 times longer. In addition to this temporal cost, the UI approach exhibited a considerably higher standard deviation of $\pm 56.8s$ and required native framework interventions (retries) in 35% of its executions to prevent a pipeline failure.

The additional time consumed by the UI test can be largely attributed to computational factors unrelated to business logic validation: network latency for downloading frontend assets (HTML, CSS, JavaScript), the browser rendering engine hydration, the execution of third-party tracking scripts, and the mandatory dynamic waiting periods required for visual elements to become interactive.

Taken together, these results support that pushing functional validation down to the API layer is not merely a micro-optimization, but an important architectural consideration for maintaining a fast, scalable, and stable feedback loop. The UI layer should be preferentially reserved for validating the final visual integration, rather than serving as the primary vehicle for functional business testing.

It is important to note that this case was chosen because it allows a more direct comparison between the two layers. However, the implementation of both API and UI E2E tests here was considered necessary because they serve different purposes, and both were considered important. However, in scenarios where the user journey or visual impact is not critical, the main takeaway from this analysis is that API testing should be prioritized to achieve faster and more reliable feedback, while UI E2E testing should be used strategically to validate critical user interactions and visual elements.

## 6.3 Asynchrony and Test Retries

The transition from monolithic applications to distributed microservices means that many business operations no longer happen instantaneously. When analyzing the execution times and standard deviations in Table 6.2, it becomes clear that both the API and UI E2E testing layers had to accommodate these asynchronous delays, even though in different ways.

Within the UI layer, tests are particularly vulnerable to execution instability. A browser automation test depends on a multitude of factors aligning correctly: network requests must complete, external scripts must load, and the visual elements must be fully rendered and actionable. When any of these factors experience a slight delay, the test step may fail. To mitigate this inherent fragility, the Cypress framework was configured to automatically retry failed test blocks. This native retry mechanism is clearly visible in the *Checkout Standard* scenario, which required framework interventions in 35% of its executions. When a failure occurred, Cypress restarted the test attempt, which contributes to the inflated average execution time (109.9s) and the high standard deviation ($\pm 56.8s$). The observed variance is consistent with test instability and the computational cost of restarting the browser flow.

Conversely, testing at the API layer eliminates the unpredictability of the browser, but it must still account for the time required by backend systems to process information and also some latency (because tests are executed in a cloud environment). For example, when an API request is made to create an order, the initial response simply confirms that the request was received. However, the business flow is not complete until the downstream systems (such as VTEX, logistics, etc) finish processing the order, logistic information, and generating the invoice. Thus, the retry mechanism that Cypress provides was also utilized in the API tests.

To validate these complex flows, scenarios like *Full Flow Order* and *Stock Update* could not rely on a single, synchronous assertion. Instead, the tests were programmed to repeatedly query the relevant endpoints at specific intervals until the desired business state was achieved or a timeout was reached. This polling-based verification loop is a plausible explanation for why the *Full Flow Order* scenario presents an average execution time of 110.0s and a high standard deviation of $\pm 58.8s$.

Unlike the UI layer, the high standard deviation observed in these API scenarios does not appear to reflect test fragility or framework failures — as evidenced by the low intervention rates of 0% to 5%. Rather, it likely reflects the natural fluctuation in backend processing times, with the test suite waiting for the cloud infrastructure to complete its internal tasks. This approach allows the automated tests to remain adaptive to real-time system performance, avoiding the brittleness associated with static, hard-coded waiting periods.

# Chapter 7

# Conclusion

This dissertation addressed the challenge of validating cross-service integrations and end-to-end business flows in a proprietary SaaS e-commerce ecosystem. In particular, the work focused on the VTEX platform, where development teams must operate under several practical constraints, including limited access to the underlying infrastructure, asynchronous processing, shared cloud environments, and dependence on external enterprise systems.

To address this problem, a hybrid automated testing strategy was designed, implemented, and evaluated in the context of a real-world e-commerce operation. The proposed approach combines API-level tests with selective end-to-end UI tests, supported by reusable helper abstractions for test data preparation, teardown, polling, and validation of asynchronous business states. The objective was not only to test important business scenarios, but also to determine how automated testing can remain useful and reliable when applied to distributed cloud systems that are partially opaque and externally controlled.

## 7.1 Synthesis of Findings

The results presented in Chapter 6 show that the proposed testing strategy is viable for validating critical business flows in a constrained SaaS environment. Despite the limitations imposed by the VTEX platform and by the surrounding cloud-based infrastructure, the implemented suite was able to exercise relevant business scenarios and provide meaningful feedback about the correctness and stability of the system under test.

A central finding of this work is that reliability in this context cannot be achieved simply by adding more end-to-end tests. Instead, automated tests must be carefully positioned at the most appropriate layer of the system. The evaluation reinforced the importance of prioritizing API-level validation whenever possible, since API tests provided faster and more deterministic feedback while still allowing complex business flows to be validated across service boundaries. This confirms that the Testing Pyramid remains highly relevant in SaaS-based microservice environments, but also shows that its practical importance increases when the UI layer is affected by browser rendering, third-party scripts, asynchronous loading, and network variability.

At the same time, the study confirms that UI-based end-to-end tests remain necessary. Their value lies in validating complete user-facing journeys and ensuring that the final interaction between the storefront, checkout, platform services, and external integrations behaves as expected from the user's perspective. However, the observed instability and retry requirements indicate that UI E2E tests should be used selectively, mainly for critical journeys where visual interaction and frontend integration are essential. They should not be treated as the primary mechanism for trigger and validating business logic.

Another important conclusion is that asynchrony must be treated as an explicit test design concern. In the evaluated system, several business operations depended on background processing, propagation delays, and eventual consistency across multiple services. For this reason, static waiting times would be insufficient and potentially brittle. The use of polling-based validation allowed the test suite to adapt to the real processing time of the underlying systems, improving robustness without introducing unnecessary delays when operations completed quickly.

The exploratory analysis of CDCT also revealed an important practical limitation. Although CDCT is valuable in many microservice architectures, its applicability becomes problematic when provider services cannot be executed locally and when the platform does not expose sufficient infrastructure control. In this context, implementing contract verification would require artificial mechanisms to prepare provider-side state, increasing complexity and potentially polluting the production codebase. Therefore, for the evaluated scenario, direct API-driven integration testing based on real business flows proved to be a more pragmatic and maintainable solution.

Overall, this dissertation shows that effective testing in proprietary SaaS ecosystems requires more than the direct application of conventional testing techniques. It requires adapting those techniques to the operational constraints of the platform, accepting that some sources of nondeterminism cannot be eliminated, and designing the test architecture to manage that uncertainty in a controlled and repeatable way.

## 7.2 Answers to Research Questions

This work was guided by three research questions, which are answered below based on the implementation and evaluation of the proposed testing strategy.

**RQ1: How can automated testing be engineered to mitigate the inherent constraints of cloud-only microservices environments?**

Automated testing in cloud-only microservice environments can be made more reliable by introducing architectural abstractions that isolate tests from environmental instability and platform constraints. In this dissertation, this was achieved through a helper layer responsible for test data management, API-driven setup and teardown operations, and reusable polling mechanisms.

The use of API-driven teardown procedures helped mitigate shared-state pollution in the development environment, ensuring that each execution started from a more predictable baseline.

This was particularly important because the tests were executed against a real cloud workspace where direct database access and full infrastructure control were unavailable.

Additionally, recursive polling mechanisms were used to validate asynchronous business states without relying on fixed waiting times. This allowed the test suite to accommodate eventual consistency and variable backend processing times while avoiding unnecessary delays. Therefore, the results indicate that cloud-only testing requires tests to be designed as adaptive workflows rather than as simple sequences of immediate assertions.

**RQ2: What are the quantitative trade-offs in execution time, standard deviation, and test stability when validating macroscopic business flows via API-level integration tests compared to UI end-to-end tests?**

The evaluation demonstrated that API-level integration tests provide substantially faster and more stable feedback than equivalent UI-based end-to-end tests when the objective is to validate business logic or cross-service processing. API tests avoid the overhead associated with browser automation, frontend rendering, DOM hydration, and third-party script loading, making them more suitable for frequent execution in a CI pipeline.

In contrast, UI E2E tests showed greater temporal variability and required more framework-level retries. This does not mean that UI tests are unnecessary, but it confirms that they are more expensive and more fragile. Their use should therefore be reserved for scenarios where validating the actual user interaction is essential, such as checkout flows, registration flows, or other critical customer journeys.

The main conclusion is that API and UI E2E tests should not be seen as interchangeable alternatives. They serve different purposes. API tests are better suited for validating functional business behavior across services, while UI tests are better suited for validating the final user-facing integration. A balanced testing strategy should combine both, but with most functional validation pushed toward the API layer, even if it is technically feasible for an UI E2E test to exercise or trigger the same business flow.

**RQ3: To what extent is Consumer-Driven Contract Testing (CDCT) viable and beneficial for ensuring interface compatibility in a serverless, proprietary cloud ecosystem where local infrastructure control is highly restricted?**

The exploratory evaluation of CDCT suggests that its viability is limited in the specific context studied. Consumer-Driven Contract Testing is conceptually useful because it allows consumers and providers to validate interface compatibility independently. However, its practical adoption depends on the ability to control provider execution and prepare provider-side state in a repeatable way.

In a proprietary SaaS ecosystem such as VTEX, these assumptions do not fully hold. Provider services cannot be executed locally, and the internal platform state cannot be directly controlled by the development team. As a result, contract verification would require additional artificial

endpoints or other indirect mechanisms to inject test data into the managed environment. This would increase maintenance effort and introduce implementation code whose only purpose is to support testing.

For this reason, CDCT was found to be less suitable than direct API-driven testing in the evaluated context. While CDCT should not be dismissed as a general technique, this work indicates that its benefits are significantly reduced when infrastructure control is limited and when provider verification cannot be performed in an isolated or locally reproducible environment.

## 7.3 Main Contributions

This dissertation contributes to both the practical and academic understanding of automated testing in proprietary SaaS-based e-commerce systems. The main contributions are the following:

- **A hybrid testing strategy for proprietary SaaS ecosystems:** The work presents a practical testing architecture that combines API-level integration tests with selective UI end-to-end tests, adapted to the constraints of the VTEX platform and a real-world e-commerce operation.
- **Reusable test engineering patterns for asynchronous systems:** The implementation demonstrates how polling-based validation, API-driven setup, and idempotent teardown procedures can improve test reliability in environments affected by eventual consistency and shared cloud state.
- **Empirical evidence on API and UI E2E testing trade-offs:** The evaluation provides concrete evidence that API-level validation offers faster and more deterministic feedback than UI-based validation for business logic, while still recognizing the importance of UI E2E tests for critical user journeys.
- **A practical assessment of CDCT in a constrained SaaS context:** The dissertation identifies limitations in applying Consumer-Driven Contract Testing when provider services cannot be executed locally and infrastructure control is restricted.
- **A real-world case study of testing in an enterprise e-commerce environment:** Unlike purely theoretical evaluations, this work was developed and assessed in the context of an operational system with real integrations, platform constraints, and business-critical workflows. Moreover, the complexity of this dissertation was not only the implementation of the tests, but more importantly the comprehension of the underlying system,. A good understanding of the system was necessary to design the tests, and to understand the results. This makes the contribution more relevant for practitioners facing similar challenges in real-world SaaS environments.

## 7.4 Future Work

Although the proposed testing strategy achieved its main objectives, several opportunities remain for future improvement and research.

First, future work could explore more advanced synthetic test data generation mechanisms. The current strategy relies on API-driven teardown procedures to reduce shared-state pollution. While effective, this approach could be further improved by generating isolated entities for each test execution, reducing the need for cleanup and improving support for parallel execution.

Second, the test suite could be extended with performance and load testing capabilities. This dissertation focused primarily on functional correctness and execution stability. However, the same business flows could be evaluated under higher load conditions to understand how asynchronous delays, retries, and eventual consistency behave during periods of increased traffic, such as promotional campaigns or peak sales events.

Third, future work could investigate the generalizability of the proposed testing patterns across other proprietary e-commerce platforms. Although this study focused on VTEX, similar constraints may exist in other SaaS-based commerce ecosystems. Applying the same principles in different environments would help determine which findings are platform-specific and which are broadly applicable.

Finally, further research could investigate alternative approaches to contract and interface validation in SaaS environments where traditional CDCT is difficult to apply. This could include exploring and architecting ways to achieve provider-side verification in a lightweight manner, by experimenting and comparing different approaches.

## 7.5 Final Remarks

This dissertation demonstrates that testing microservice-oriented systems within proprietary e-commerce SaaS platforms is both necessary and challenging. Although the evaluated environment does not correspond to a fully self-managed microservices architecture, it exhibits many of the same testing difficulties: distributed service interactions, asynchronous processing, eventual consistency, dependency on external systems, and limited visibility over internal execution. In the VTEX context, these challenges are further intensified by the proprietary nature of the platform, where development teams have restricted control over infrastructure, runtime behavior, and internal service boundaries.

The main lesson of this work is that effective test automation in this context requires a pragmatic integration and system-level strategy. API-level tests should be used to validate most cross-service business logic and integration behavior, since they provide faster and more deterministic feedback while avoiding the instability introduced by browser automation. UI end-to-end tests remain important, but should be reserved for critical user-facing journeys where the complete interaction between the storefront, checkout, VTEX services, and external integrations must be

validated from the user's perspective. Similarly, asynchronous operations should be validated through adaptive mechanisms, such as polling, rather than through static waits.

The study also shows that techniques commonly associated with microservices testing, such as Consumer-Driven Contract Testing, cannot be adopted uncritically in proprietary SaaS ecosystems. Their usefulness depends on the degree of control available over provider services, execution environments, and test data preparation. In environments such as VTEX, where these conditions are only partially available, direct integration and system-level validation may provide a more practical and maintainable path to confidence.

By applying these principles, the proposed strategy provides a structured way to increase confidence in complex e-commerce integrations while respecting the limitations of a cloud-only, proprietary SaaS environment. Although the approach does not eliminate the inherent uncertainty of distributed systems, it offers a maintainable foundation for continuously validating business-critical flows in real-world enterprise software development.

# References


[1] Amr S. Abdelfattah et al. “Test Coverage in Microservice Systems: An Automated Approach to E2E and API Test Coverage Metrics”. In: *Electronics (Switzerland)* 13.10 (2024), p. 1913. ISSN: 2079-9292. DOI: 10.3390/electronics13101913.

[2] Hamdy Michael Ayas et al. “An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing”. In: *Proceedings - 48th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2022*. IEEE Computer Society, 2022, pp. 92–99. ISBN: 9781665461528. DOI: 10.1109/SEAA56994.2022.00022.

[3] Sebastian Bello-Trejo et al. “System-Oriented Testing on the Microservices Architecture: A Systematic Literature Review”. In: *Proceedings - 2024 12th International Conference in Software Engineering Research and Innovation, CONISOFT 2024*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 127–136. ISBN: 9798331532116. DOI: 10.1109/CONISOFT63288.2024.00026.

[4] Matteo Camilli et al. “Microservices Integrated Performance and Reliability Testing”. In: *Proceedings - 3rd ACM/IEEE International Conference on Automation of Software Test, AST 2022*. Institute of Electrical and Electronics Engineers Inc., July 2022, pp. 29–39. ISBN: 9781450392860. DOI: 10.1145/3524481.3527233.

[5] Jeremy J. Carroll, Pankaj Anand, and David Guo. “Preproduction Deploys: Cloud-Native Integration Testing”. In: *2021 IEEE Cloud Summit*. 2021, pp. 41–48. DOI: 10.1109/IEEECloudSummit52029.2021.00015.

[6] Boni García et al. “Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright”. In: *Communications in Computer and Information Science*. Vol. 2178 CCIS. Springer Science and Business Media Deutschland GmbH, 2024, pp. 142–149. ISBN: 9783031702440. DOI: 10.1007/978-3-031-70245-7_10.

[7] Luca Giamattei et al. “Assessing Black-box Test Case Generation Techniques for Microservices”. In: *Communications in Computer and Information Science*. Vol. 1621 CCIS. Springer Science and Business Media Deutschland GmbH, 2022, pp. 46–60. ISBN: 9783031141782. DOI: 10.1007/978-3-031-14179-9_4.

[8] Jiayu Gong and Lizhi Cai. “Analysis for Microservice Architecture Application Quality Model and Testing Method”. In: *2023 26th ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD-Winter 2023*. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 141–145. ISBN: 9798350345865. DOI: 10.1109/SNPD-Winter57765.2023.10223960.

[9] Cristian Martínez Hernández et al. “Comparison of End-to-End Testing Tools for Microservices: A Case Study”. In: 2021, pp. 407–416. DOI: 10.1007/978-3-030-68285-9_39.

[10] Mingxuan Hui et al. “Unveiling the microservices testing methods, challenges, solutions, and solutions gaps: A systematic mapping study”. In: *Journal of Systems and Software* 220 (Feb. 2025). ISSN: 01641212. DOI: 10.1016/j.jss.2024.112232.

[11] Waleed Ibrahim and Van Kiet Luong. “Transforming E-commerce: A Microservices-Based Taxonomy for Cloud Applications”. In: *Lecture Notes in Networks and Systems*. Vol. 1346 LNNS. Springer Science and Business Media Deutschland GmbH, 2025, pp. 476–486. ISBN: 9783031876462. DOI: 10.1007/978-3-031-87647-9_39.

[12] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 3.0*. Los Alamitos, CA, USA: IEEE, 2014. ISBN: 978-0-7695-5166-1.

[13] Mariam Kiran and Anthony J. H. Simons. “Testing Software Services in Cloud Ecosystems”. In: *International Journal of Cloud Applications and Computing* 6.1 (2016), pp. 42–58. DOI: 10.4018/IJCAC.2016010103.

[14] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. “Consumer-Driven Contract Tests for Microservices: A Case Study”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11915 LNCS. Springer, 2019, pp. 497–512. ISBN: 9783030353322. DOI: 10.1007/978-3-030-35333-9_35.

[15] Hongwei Li et al. “Research on Microservice Application Testing System”. In: *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*. 2020, pp. 363–368. DOI: 10.1109/ICISCAE51034.2020.9236829.

[16] Tingshuo Miao, Asif Imtiaz Shaafi, and Eunjee Song. *Systematic Mapping Study of Test Generation for Microservices: Approaches, Challenges, and Impact on System Quality*. Apr. 2025. DOI: 10.3390/electronics14071397.

[17] Kiet Ngo et al. “Research on Test Flakiness: from Unit to System Testing”. In: (2022). DOI: 10.1145/3551349.

[18] Pact Foundation. *Pact Documentation*. https://docs.pact.io/. Accessed: 2025-01-05. 2025.

[19] Pactflow. *Pactflow Documentation*. Accessed: 2025-01-05. 2025. URL: https://docs.pactflow.io/.

[20] Francisco Ponce et al. “Microservices testing: A systematic literature review”. In: *Information and Software Technology* 188 (2025), p. 107870. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2025.107870.

[21] Fábio Santos and Ricardo Martinho. “Architectural Challenges on the Integration of e-Commerce and ERP Systems: A Case Study”. In: *International Conference on Enterprise Information Systems, ICEIS - Proceedings*. Vol. 1. Science and Technology Publications, Lda, 2021, pp. 313–319. ISBN: 9789897585098. DOI: 10.5220/0010494903130319.

[22] Georg Daniel Schwarz, Felix Quast, and Dirk Riehle. “Ensuring Syntactic Interoperability Using Consumer-Driven Contract Testing”. In: *Software Testing Verification and Reliability* 35 (5 Aug. 2025). ISSN: 10991689. DOI: 10.1002/stvr.70006.

[23] Manuel Simosa and Frank Siqueira. “Contract Testing in Microservices-Based Systems: A Survey”. In: *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*. IEEE Computer Society, 2023, pp. 312–317. ISBN: 9798350336269. DOI: 10.1109/ICSESS58500.2023.10293058.

[24] Stelios Sotiriadis et al. “Unit and integration testing of modular cloud services”. In: *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*. Institute of Electrical and Electronics Engineers Inc., May 2017, pp. 1116–1123. ISBN: 9781509060283. DOI: 10.1109/AINA.2017.57.

[25] VTEX Developers. *Composability in VTEX*. Accessed: 2026-01-05. 2026. URL: https://developers.vtex.com/docs/guides/composability.

[26] Muhammad Waseem et al. “Testing microservices architecture-based applications: A systematic mapping study”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. Vol. 2020-December. IEEE Computer Society, Dec. 2020, pp. 119–128. ISBN: 9781728195537. DOI: 10.1109/APSEC51365.2020.00020.

[27] Chu Fei Wu et al. “Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. Vol. 2022-December. IEEE Computer Society, 2022, pp. 467–471. ISBN: 9781665455374. DOI: 10.1109/APSEC57359.2022.00064.

[28] Blerand Zendeli et al. “Identifying Limitations in End-to-End Testing: A Systematic Review of Coverage Analysis Techniques”. In: *ICHORA 2025 - 2025 7th International Congress on Human-Computer Interaction, Optimization and Robotic Applications, Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2025. ISBN: 9798331510886. DOI: 10.1109/ICHORA65333.2025.11016980.

# Appendix A

# Test Execution Results

### Pass/Fail/Retried

| Scenario | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Run 6 | Run 7 | Run 8 | Run 9 | Run 10 | Run 11 | Run 12 | Run 13 | Run 14 | Run 15 | Run 16 | Run 17 | Run 18 | Run 19 | Run 20 | Success | Interventions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Full Flow order | V | V | V | V | V | V | V | V | V | V | R | V | V | V | V | V | V | F | V | V | 95,00% | 5,00% |
| Order multibanco | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | 100,00% | 0,00% |
| Stock update | R | V | V | V | V | V | V | V | V | V | V | V | R | R | V | V | V | V | V | V | 100,00% | 15,00% |
| XML Mainland | V | V | V | V | V | R | V | V | V | V | R | V | V | V | V | V | V | V | V | V | 100,00% | 10,00% |
| XML Islands | V | V | V | V | V | V | V | V | V | V | V | V | V | R | V | V | V | V | V | V | 100,00% | 5,00% |
| XML pickup | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | 100,00% | 0,00% |
| Customer Creation | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | 100,00% | 0,00% |
| Customer Update | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | 100,00% | 0,00% |
| Checkout standard | R | R | R | R | V | F | R | R | V | V | R | V | V | V | V | V | V | V | V | V | 95,00% | 35,00% |
| Checkout pickup | V | R | R | V | V | V | V | R | V | V | V | V | V | V | R | V | V | V | V | V | 100,00% | 20,00% |
| Machine Registration | V | R | V | V | V | V | V | V | V | V | V | V | V | V | V | V | V | R | V | V | 100,00% | 10,00% |

V - Passed
R - Cypress retried the test
F - Failed

### Execution Time

| Scenario | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Run 6 | Run 7 | Run 8 | Run 9 | Run 10 | Run 11 | Run 12 | Run 13 | Run 14 | Run 15 | Run 16 | Run 17 | Run 18 | Run 19 | Run 20 | Average | STD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Full Flow order | 86 | 70 | 107 | 88 | 60 | 59 | 96 | 96 | 103 | 96 | 214 | 116 | 81 | 103 | 106 | 99 | 117 | 321 | 87 | 96 | 110,05 | 58,75056551 |
| Order multibanco | 21 | 13 | 12 | 11 | 11 | 16 | 11 | 12 | 12 | 11 | 12 | 11 | 14 | 12 | 16 | 13 | 16 | 12 | 12 | 14 | 13,1 | 2,51102830 |
| Stock update | 189 | 135 | 139 | 133 | 136 | 137 | 137 | 135 | 137 | 140 | 136 | 137 | 209 | 193 | 135 | 137 | 136 | 135 | 139 | 137 | 145,6 | 22,47431282 |
| XML Mainland | 83 | 77 | 78 | 79 | 77 | 84 | 77 | 81 | 78 | 75 | 142 | 91 | 82 | 79 | 86 | 90 | 88 | 83 | 95 | 82 | 85,35 | 14,36836804 |
| XML Islands | 78 | 76 | 76 | 77 | 76 | 75 | 77 | 76 | 75 | 75 | 75 | 80 | 82 | 145 | 85 | 86 | 87 | 76 | 85 | 83 | 82,25 | 15,34472308 |
| XML pickup | 30 | 28 | 28 | 30 | 26 | 28 | 27 | 27 | 31 | 29 | 36 | 28 | 28 | 26 | 28 | 28 | 28 | 25 | 25 | 27 | 28,15 | 2,412140347 |
| Customer Creation | 12 | 8 | 5 | 11 | 10 | 8 | 8 | 12 | 8 | 8 | 4 | 16 | 5 | 5 | 8 | 4 | 5 | 16 | 8 | 4 | 8,25 | 3,668858994 |
| Customer Update | 28 | 16 | 14 | 4 | 10 | 9 | 7 | 10 | 10 | 16 | 10 | 8 | 13 | 13 | 17 | 11 | 8 | 11 | 4 | 5 | 11,2 | 5,463756179 |
| Checkout standard | 144 | 139 | 152 | 207 | 67 | 205 | 162 | 228 | 65 | 72 | 142 | 67 | 68 | 55 | 63 | 79 | 73 | 76 | 68 | 66 | 109,9 | 56,75329807 |
| Checkout pickup | 87 | 149 | 149 | 61 | 58 | 64 | 56 | 218 | 66 | 62 | 67 | 55 | 58 | 56 | 115 | 97 | 64 | 59 | 92 | 61 | 84,7 | 42,88307236 |
| Machine Registration | 156 | 219 | 135 | 120 | 120 | 168 | 147 | 146 | 164 | 142 | 107 | 130 | 117 | 123 | 124 | 142 | 117 | 171 | 146 | 136 | 141,5 | 25,68534294 |
| Total | 914 | 930 | 895 | 821 | 651 | 853 | 805 | 1041 | 749 | 726 | 945 | 739 | 757 | 810 | 783 | 786 | 739 | 985 | 761 | 711 | 820,05 | 101,8468794 |