nagarro

Conquering complexity: Test Automation for multi-brand omnichannel e-commerce

by **Bimal Tissakuttige**

Table of Contents

E-commerce business: An overview 01

Bridging the gaps: Critical testing needs for modern e-commerce 03

The real challenges in test automation 07

Optimizing your testing efforts: A smarter approach to testing 09

A strategic approach to automated testing 11

1. Tool selection 11

2. Prioritization of test cases 12

3. Maintenance and scalability 14

4. Test early and often 17

5. Test with speed 18

6. Process 19

7. Accelerators 21

7.1 Re-running of tests 21

7.2 Error classification report 22

7.3 Dashboards 23

7.4 Self-healing and prevention 24

7.5 Test case vs Test script sync 25

8. Visual regression 25

9. Collaboration and communication 26

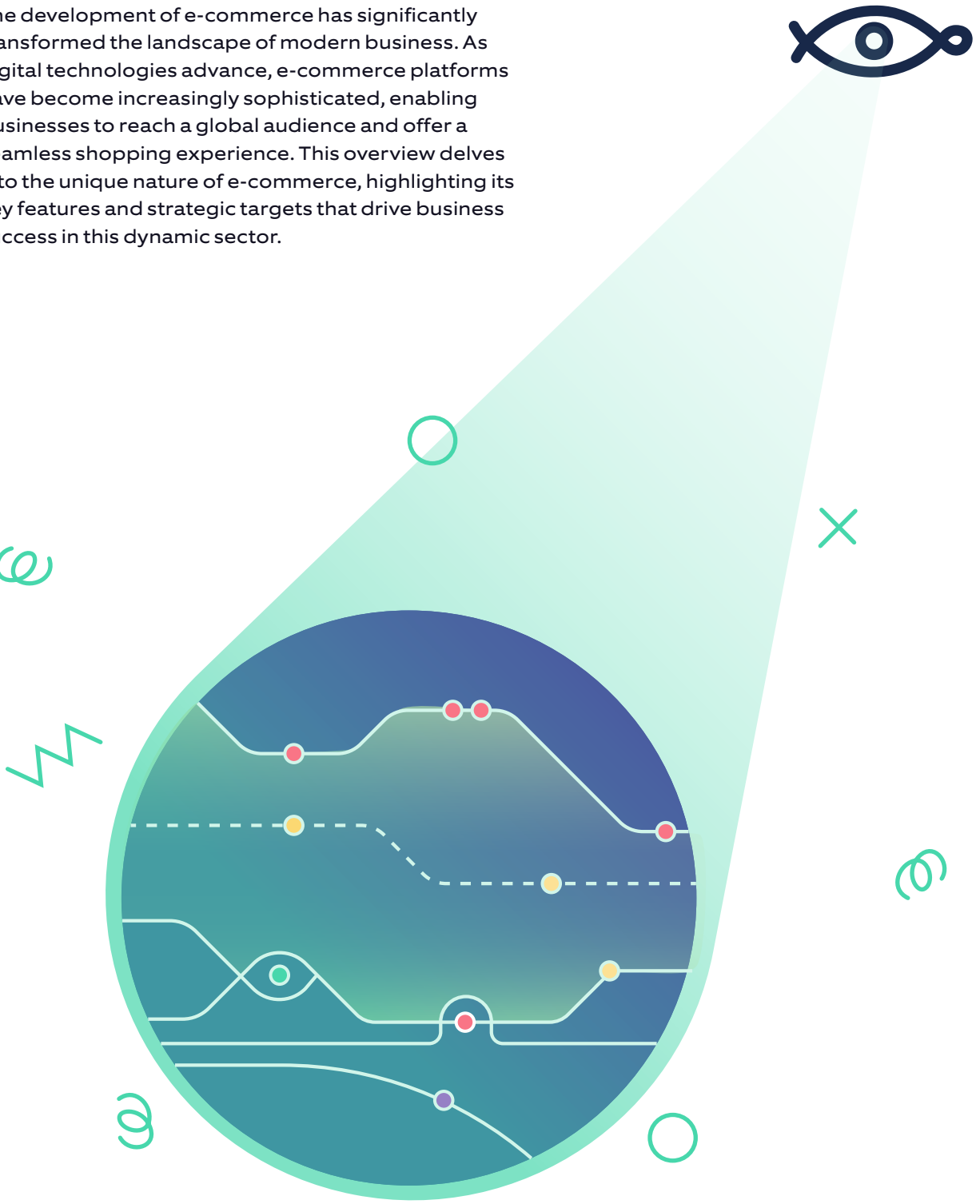
Conclusion: mastering omnichannel e-commerce test automation 27

Glossary of Abbreviations 27

About the author 27

E-commerce business: An overview

The development of e-commerce has significantly transformed the landscape of modern business. As digital technologies advance, e-commerce platforms have become increasingly sophisticated, enabling businesses to reach a global audience and offer a seamless shopping experience. This overview delves into the unique nature of e-commerce, highlighting its key features and strategic targets that drive business success in this dynamic sector.



Unique nature

1. Omnichannel integration

- Seamless customer experience: Customers can switch between online and offline channels seamlessly. For instance, they might browse products online and purchase in-store, or vice versa.
- Unified inventory management: Integrates inventory across all sales channels, ensuring better stock visibility and management.
- Consistent branding and messaging: Ensures that customers receive a consistent brand experience regardless of the channel they use.

2. Multi-regional capabilities

- Localization: Websites can offer content in multiple languages, accept different currencies, and adhere to regional legal requirements.
- Regional warehousing and logistics: Setting up regional warehouses to reduce shipping times and costs.
- Market-specific strategies: Tailoring marketing and promotional strategies to suit regional preferences and behaviors.

Targets

1. Increased reach and market penetration

- Global access: Ability to tap into new markets by making the website accessible to a broader audience.
- 24/7 availability: Customers can shop anytime, breaking the limitations of physical store hours.

2. Enhanced customer experience

- Personalization: Use data from various channels to provide personalized recommendations and offers.
- Convenience: Providing multiple purchasing options, such as buy online, pick up in-store (BOPIS), enhances customer convenience.

3. Operational efficiency

- Integrated systems: Streamlining operations by integrating sales, inventory, and customer service systems across all channels.
- Data analytics: Leveraging data from all regions and channels to make informed business decisions.

4. Competitive advantage

- Adaptability: Being able to quickly adapt to market changes and customer preferences in different regions.
- Brand loyalty: Providing a consistent and high-quality customer experience builds brand loyalty across different regions.

Bridging the gaps: Critical testing needs for modern e-commerce

As the e-commerce landscape evolves, businesses are expanding their reach by adopting omnichannel strategies and serving multiple regions with localized versions of their platforms. This complexity brings a unique set of challenges to test automation, especially when dealing with multiple brands and internationalization. Here, we will explore the key challenges, a structured test approach, and an effective strategy for automating tests in such environments.

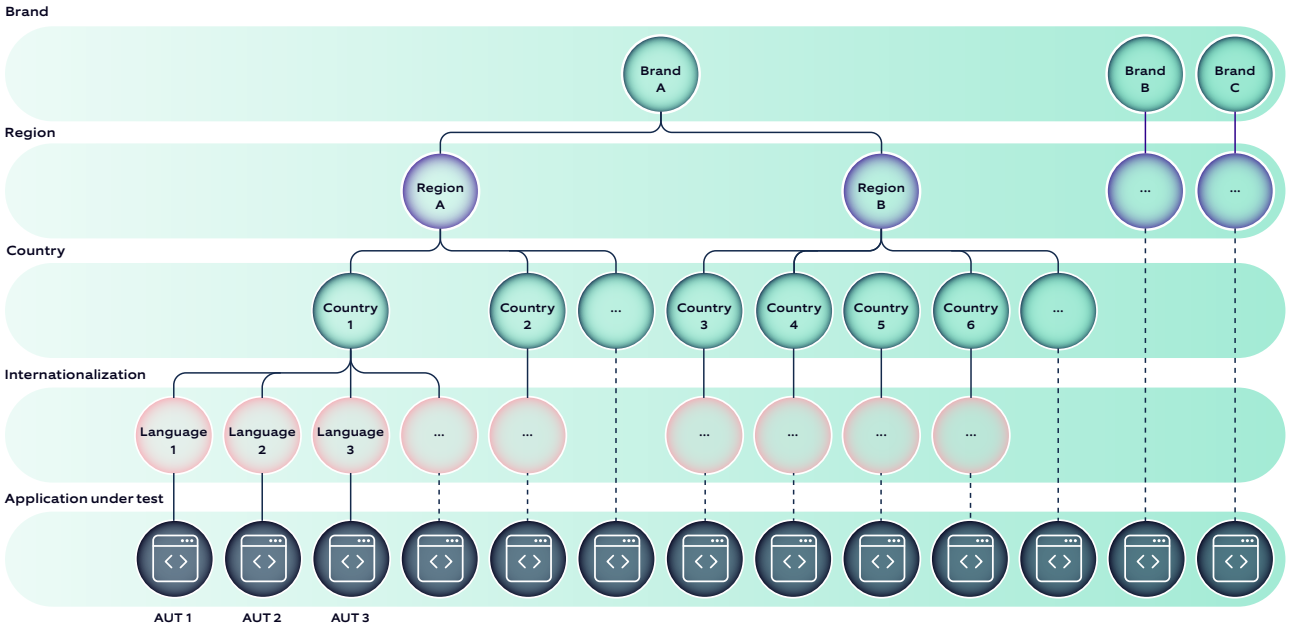
In this article, we will focus on the **end-to-end** testing aspect of e-commerce application test automation, as the entire test automation strategy is a vast topic encompassing many different levels of testing automation.

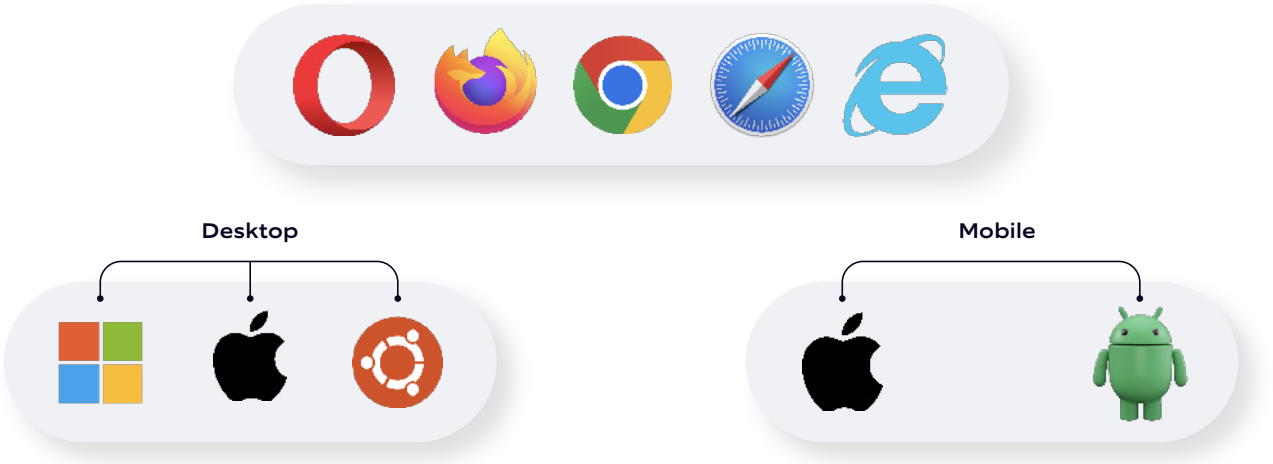
In this article, we will illustrate how the test automation principles are applied in real-world scenarios by providing specific examples and references through the following use case. You might notice that we use specific terms instead of more general ones. For example, we'll mention 'Zephyr' as the tool, rather than using the broader term 'Test Management tool.'

Use case example: Global fashion retailer

The following use case is inspired by an actual project; however, details have been anonymized to ensure the privacy and confidentiality of the client are fully protected.

A global apparel and footwear company operates multiple e-commerce websites for different brands (e.g., Brand A, Brand B, and Brand C) and serves customers across various regions (e.g., North America and Europe). Each region comprises multiple countries, with each country supporting multiple languages. Each brand has a unique design and user experience, and the websites need to support multiple languages, currencies, and localized content. The websites cater to users of web browsers on all major desktop and mobile operating systems.





The websites are developed using a common UI platform or different platforms, depending on the brand's choice, using Vue.js, while the backend is developed using RESTful APIs based on the Mulesoft API platform. The CMS is CoreMedia, and the CRM is Salesforce. The websites use various payment methods and gateways such as Credit Card, Gift Cards, PayPal, Klarna, Apple Pay, Adyen, and Cybersource.

The development teams practice two-week sprints, and the applications are released to production at the end of every sprint. The project management tool used is JIRA, and the test management tool is Zephyr.

Functional testing is carried out covering all the above application instances/contexts in many different development, test (QA, staging), and production environments.

- Total regression test count per brand/country/region/language/environment: **1300**
- Total number of AUT instances to be tested: **16 (assuming 1 language per country)**
- Total number of regression tests: **1300 * 16 = 20800** (assuming test count per instance are all equal, though realistically, it slightly varies)

The real challenges in end-to-end test automation

As e-commerce platforms evolve, the complexities of maintaining a seamless, high-quality user experience across multiple brands and regions become increasingly pronounced. These challenges are compounded when businesses adopt omnichannel strategies and need to cater to a diverse global audience. Test automation, while a powerful tool for ensuring consistency and efficiency, faces significant hurdles in such multifaceted environments. In this section, we will delve into the primary test automation challenges that arise.

Diverse user experiences

Each brand has its own identity and user experience, which needs to be maintained across different regions. Variations in UI/UX design across brands and regions add complexity to test automation scripts.

Example: Brand A has a minimalist design while Brand B uses a more vibrant, image-heavy layout. Test scripts must handle these UI differences.

Internationalization (i18n) and localization (l10n)

Support for multiple languages, currencies, and regional settings requires comprehensive testing to ensure correct functionality. Handling date formats, number formats, and text direction (e.g., left-to-right vs. right-to-left) in automated tests.

Example: Brand A website might need to support English, French, German and Italian. Automated tests must check that translations are correct and that the layout adapts to languages.

Dynamic content and personalization

Content might change based on user profiles, preferences, and regional promotions. Automated tests need to account for dynamic and personalized content, ensuring consistency.

Example: Homepages might display different promotions based on user location or past purchases. Automated tests should verify that the correct promotions are shown.

Multi-device and multi-platform testing

Ensuring a seamless experience across various devices (mobile, tablet, desktop) and platforms (iOS, Android, Web). Managing different screen sizes, resolutions, and operating system versions.

Example: Testing an app on iOS and Android devices, as well as different web browsers like Chrome, Firefox, and Safari.

Test maintenance

Certain features may apply to certain brand-region-country websites only while certain other features apply to all websites. Keeping test scripts up-to-date with evolving applications features and design changes.

Example: Subjected applications are evolving consistently, where tests identified to be automated with priority come to about 50-75 test cases per sprint on average. Additionally, 3-5% tests are removed from the scope, and 15-20% tests are changed out of the entire regression set. Test scripts should handle such frequent changes over a long period of time.

Testing demand

In addition to many different brand-region-country-language specific websites, it's expected to test multiple different versions of them simultaneously.

Example: Testing the version n on staging environment while n+1 version on QA environment and development environment.

Integration with backend systems

Integration with many different payment gateways, inventory management systems, and other third-party services must be tested thoroughly. Handling asynchronous processes and APIs in test scripts.

Example: Testing the checkout process involves verifying integrations with different payment gateways. Successful order placement needs to be verified in other integrated systems.

Optimizing your testing efforts: A smarter approach to testing

A comprehensive test approach is crucial for effectively automating tests in a complex e-commerce environment with multiple brands and regions. The following steps outline a structured approach to tackle the challenges:

Requirement analysis and test planning

Understanding the requirements for each brand and region, including unique features and regional customizations, is the first step. Create a detailed test plan that includes the scope of automation, test objectives, tools, and technologies to be used. For example, when implementing a new feature like multi-currency support, define the scope, objectives, and key scenarios to be automated.

Modular test design

Adopt a modular approach by creating reusable test components and libraries that can be shared across different brands and regions. Use parameterized tests to handle different regional settings and configurations. For instance, create reusable modules for login, product search, and checkout that can be used across different brands.

Test data management

Manage test data effectively to support different languages, currencies, and regional settings. Use data-driven testing techniques to validate multiple scenarios with varied inputs. An example would be using a data-driven approach to test different payment methods (credit card, PayPal, Apple Pay) across regions.

Automation framework

Choose a robust and scalable automation framework that supports multiple devices, platforms, and languages. Implement a layered architecture with separate layers for test scripts, test data, and configuration settings. For instance, use Selenium WebDriver for web automation and Appium for mobile automation within the same framework.

Continuous integration and continuous deployment (CI/CD)

Integrate test automation with CI/CD pipelines to ensure tests are run continuously and feedback is received promptly. Use containerization and virtualization techniques to replicate different environments for testing. An example would be integrating automated tests with Jenkins to run tests on every code commit and deploy to staging environments.

Test environment management

Setting up and maintaining multiple testing environments for many different application regional variations, simultaneously, is crucial. For example, use Docker to create consistent test environments that mirror required settings.

Monitoring and reporting

Implement comprehensive monitoring and reporting mechanisms to track test execution and results. Use dashboards and alerts to provide real-time feedback on test status and issues. An example would be using tools like Allure or TestRail to generate detailed test reports and track test execution status.

A strategic approach to end-to-end automated testing

Tools may be selected at the beginning or during test implementation based on requirements and improvements/changes. For example, selecting the main automation tool at the start, or adding visual regression or accessibility testing support later as an improvement.

1. Tool selection

Research

Conduct thorough research on available tools, considering key factors such as:

- Support for Vue.js and RESTful APIs
- Cost-effectiveness
- Integration with CI/CD, test management, and reporting tools
- Flexibility for future additions (visual regression, accessibility, performance testing)
- Customizability and extendability
- Scalability and ease of management

Popular tools considered

Evaluate tools based on project requirements:

- Web testing: TestCafe, Selenium, Cypress, Puppeteer
- Mobile testing: TestCafe, Appium, Espresso, XCUITest
- API testing: Postman, RestAssured, SoapUI
- CI/CD tools: Jenkins, CircleCI, GitHub Actions

Selection criteria

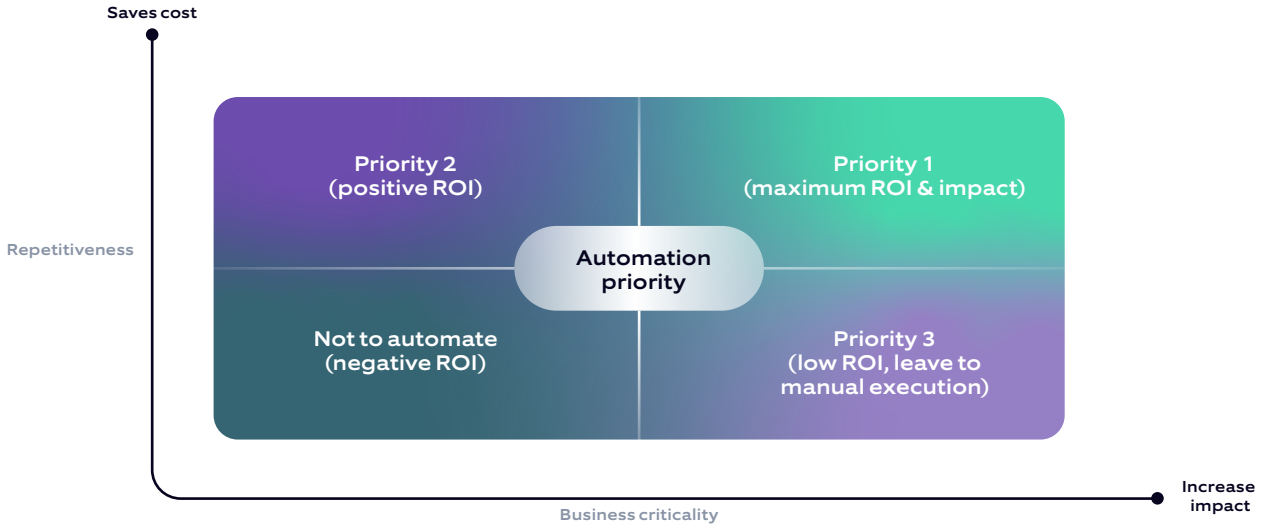
Consider the following when choosing tools:

- Match with team skills and minimal upskilling effort
- Documentation and community support
- Ease of troubleshooting and problem-solving
- Conduct POCs against project-specific requirements

In this instance, after careful consideration and POCs, the decision is made to build the test automation framework around tools and technologies such as TestCafe, TypeScript, Gherkin, Docker, Jenkins, Bitbucket, Browserstack, AWS, and Nexus.

2. Prioritization of test cases

Handling over 20,000 tests with automation can be overwhelming for any team, considering the shorter release cycles (shorter window for testing inherited with it), and limitations in capacity and infrastructure cost. To address this challenge, we aim for an approach that maximizes the effectiveness of the testing process while optimizing resources.



- Prioritize test cases based on business impact, customer requirements, business criticality, defect-prone areas, new features, compliance needs, manual execution complexity and frequency of use.
- Focus on high-value tests like smoke and regression tests, prioritizing frequently run and repetitive tests to save manual effort.
- Strongly consider testcase stability before picking them up for automation.
- Consider simple tests to be automated with priority.
- To gain faster coverage, take up data driven tests with priority whenever possible.
- Prioritize testcases considering ROI, based on cost benefit analysis and eyeing the long-term gains.
- Target to keep up with the development and manual testing without letting too many important tests spill over to the next sprint and eventually to the backlog.
- Understand that 100% test automation is not a realistic goal. Due to ROI, infeasibility and importance in performing them manually, certain tests can be left out of the priority list. Make this transparent to other stakeholders and get their buy-in.
- Mark tests with appropriate labelling that are not automated with priority for potential future automation.

- Target to achieve approximately 80% test automation coverage for the complete regression test set. When there's enough capacity available, go after the lower priority cases.
- Track testcase information such as priority (High, Medium, Low), applicability for different brand/region/country/language/devices, and test suites (smoke/regression/sanity) in Zephyr test management tool.
- Design test cases focusing on end-to-end scenarios that cover major user journeys across different channels and regions compared to having many repetitive granular level tests which may consume comparatively higher execution time.

  Example: Prioritizing checkout process tests for all brands and regions is crucial as it directly impacts revenue.

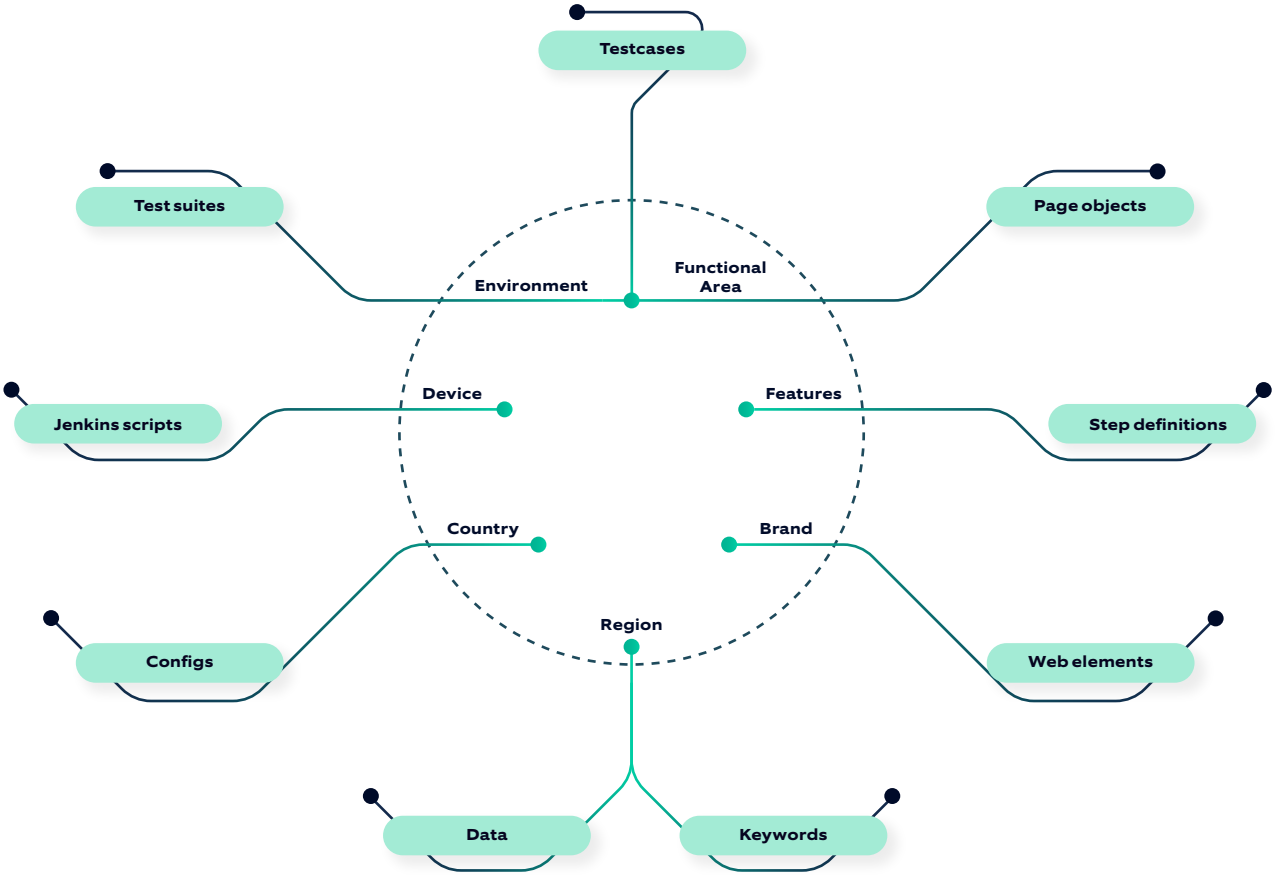
- The team's expertise should be strongly considered during assignment of tests to be automated.
- Choose which application instances the tests are to be executed on, as it's not viable to execute all test cases against all their applicable brand/region/country/devices.

  Example:

  1. Test high priority tests across all language instances while conducting the rest of the regression tests against one language per brand-region-country.
  2. Reduce multiple supported browsers specific test repetition by introducing cross-browser test suites. For instance, Google Chrome is used for all regression test executions as the primary browser while smaller cross-browser suites take care of browser compatibility tests.

3. Maintenance and scalability

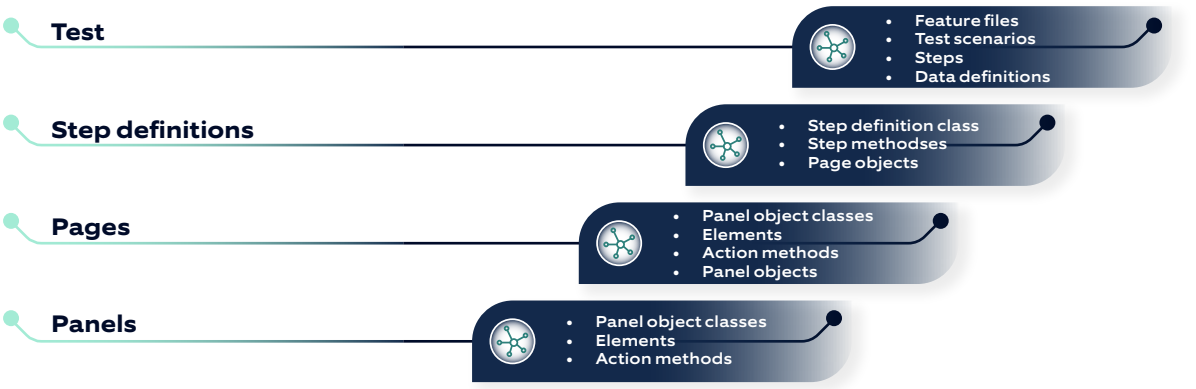
Ensuring test scripts are easily maintainable as the application evolves is crucial for long-term success. The automation framework should be designed to be scalable, allowing the addition of new brands, regions, and channels with minimal effort. To achieve this, it's important to organize main segments of the test automation framework (test cases, page objects, step definitions, web elements, keywords, data, configs, jenkins scripts, and test suites) to be configurable based on functional area, features, brand, region, device, country, language, and environment.



Framework design and organization

To support maintenance and scalability, consider the following strategies:

1. Design test cases to support applicable brand-region-country combinations by handling conditional business logic and adding specific tags at the scenario level to facilitate execution using tags/tags combinations.
2. Use device-specific tags at the test scenario level for execution against different devices.
3. Classify test cases, page objects, and web elements under functional area/feature-specific folder/file structure for team specialization.
4. Construct web element locators with conditions based on brand-region-country information.
5. Use the page object design pattern, with page objects holding elements, actions performed on those elements and panel object instances. panel objects hold repetitive/common web element sections (example: header/footer) which helps optimization of resources usage.
6. Ensure step definition methods interact with the application only through page objects' action methods.



Data management and CI/CD integration

1. Maintain test data (example: products, users, promotions, stores, payment) in brand-region-country and environment specific files so during execution tests can have different data samples based on parameters.
2. Minimize hard coding data and use data definition keywords at the feature file level. Values held by keywords are realized during runtime based on what they held in brand-region-country and environment specific data files.
3. In data files, use language-specific keys for data-level translations.
4. Integrate ci using jenkins master script and test execution-specific jenkins configs. Master script holds all the conditional logic to handle different types of test executions depending on parameters injected via config files during each execution.
5. Jenkins scripts are classified based on functional area, environment, brand-region-country, device and test suite information so it provides an easier and organized way to scale the automation with newly added brands, regions, countries and devices.

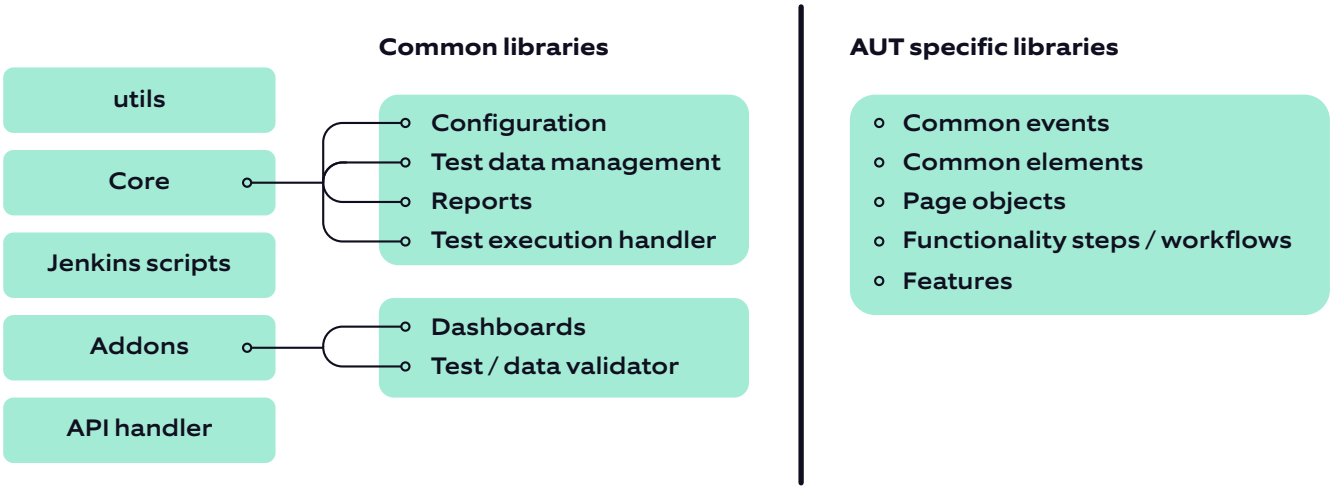
Additional best practices

Maintain 1:1 mapping between jenkins pipelines, jenkins scripts, and zephyr test plans for easier artifact creation. When a new type of test pipeline execution (new brand-region-country specific smoke test) is needed, only a pair of new jenkins config and corresponding to that is expected to be created.

Update test execution results in test cycles and present test automation coverage easily by maintaining 1:1 mapping between automation test scenarios and zephyr test case.

Use backend services/apis for setup and teardown steps to improve efficiency and stability oppose to using frontend.

Introduce dependency and package management (segregating common utility libraries which are not sensitive to changes in the application under test) for better version control (backward compatibility), improved code reusability across different source code repositories/branches and support modular development.



Use docker images for cost-effective and scalable remote test execution infrastructure.
(example: npm and required browser versions installed)

4. Test early and often

Leveraging automation to detect defects early in the application development lifecycle is crucial for maintaining high-quality e-commerce platforms. By implementing a "test early and often" approach, testing teams can provide valuable feedback on the impact of new feature development. This strategy helps development teams improve quality, mitigate risks, and increase confidence in released features.

To ensure comprehensive testing coverage, it's important to test frequently across various environments:

1. Development environment

In the development environment, smoke tests are run every 4 hours. This frequent testing helps catch issues quickly as developers make changes.

2. QA environment

The QA environment sees more extensive testing on a daily basis. This includes smoke tests, regression tests, and mobile testing to ensure thorough coverage of all aspects of the application.

3. Staging environment

For each release to the staging environment, a comprehensive suite of tests is run. This includes smoke tests, regression tests, cross-browser testing, and mobile testing. This thorough approach helps identify any issues before moving to production.

4. Production environment

In the production environment, smoke tests are run for each release. This final check helps ensure that the core functionality of the application is working as expected in the live environment.

5. Test with speed

To ensure efficient and rapid testing across multiple e-commerce applications, the following strategies are implemented:

- Opt in for parallel threaded execution and use dedicated users in each thread to avoid data dependency
- Design jenkins pipelines setup in a way that all tests for all brand-region-country-language websites can be run simultaneously for a given environment or for multiple environments, depending on the requirements.
- The complete regression test suite is broken down into functional area specific tests that allows easier work distribution with more specialized domain experts. Each functional area specific test suite is run with 5-10 parallel threads depending on the capacity of the aws ec2 instance (jenkins slave) that's assigned for tests execution. Assuming total functional area count is 4 - 5, approximately 40 tests are run parallel per website
- A round of test execution takes not more than 1.5 hours. As all tests get triggered at the same time, all test executions are completed under 2 hours
- Use browser headless mode for test execution
- Use browserstack for mobile device specific test execution and highly UI- sensitive tests execution (example: checkout flow)
- Rerunning of failed tests is automatically triggered immediately based on the execution results using jenkins pipelines (rerun pipelines are designed corresponding to original test execution pipelines) and keeps repeating based on the results of subsequent rounds of test execution.
  Rerunning of failed tests is explained in section 7.1

6. Process

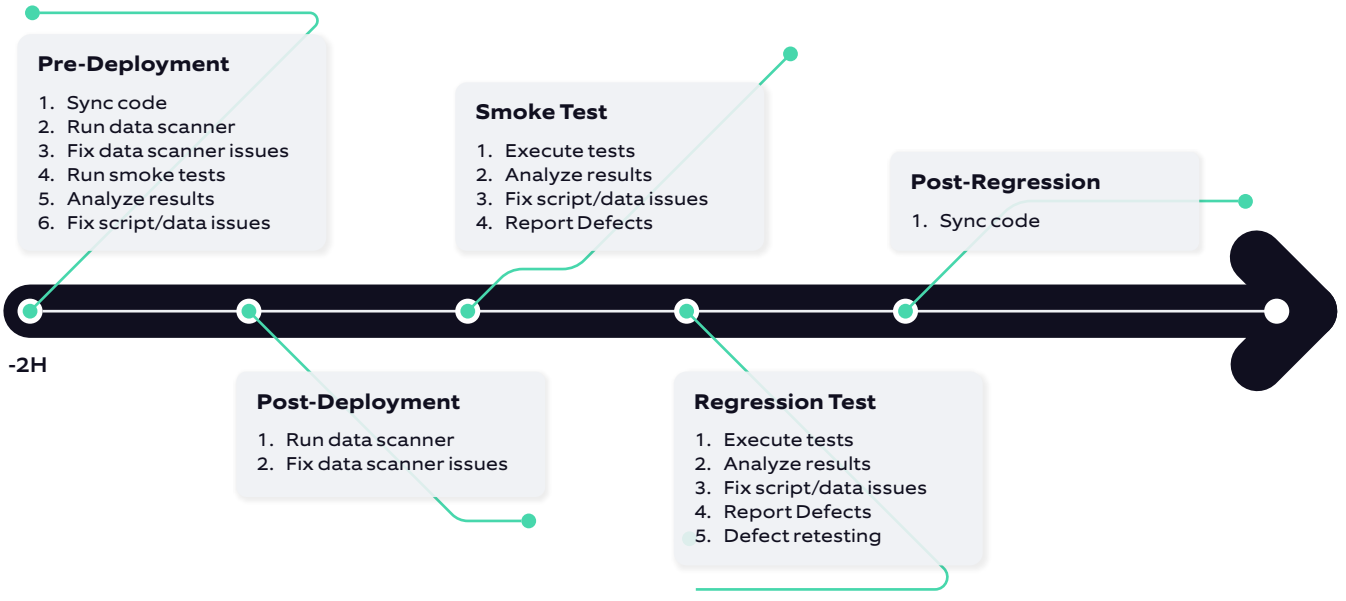
To optimize the overall development process and ensure effective test automation, it's crucial to adopt a holistic approach that goes beyond simply automating existing functional test cases. This involves:

- Contributing progressive ideas and fostering collaboration to optimize the entire development process
- Making stakeholders aware of automation's contributions, limitations, and benefits
- Understanding priorities better and negotiating risk mitigation and process improvements

Examples of process improvements include:

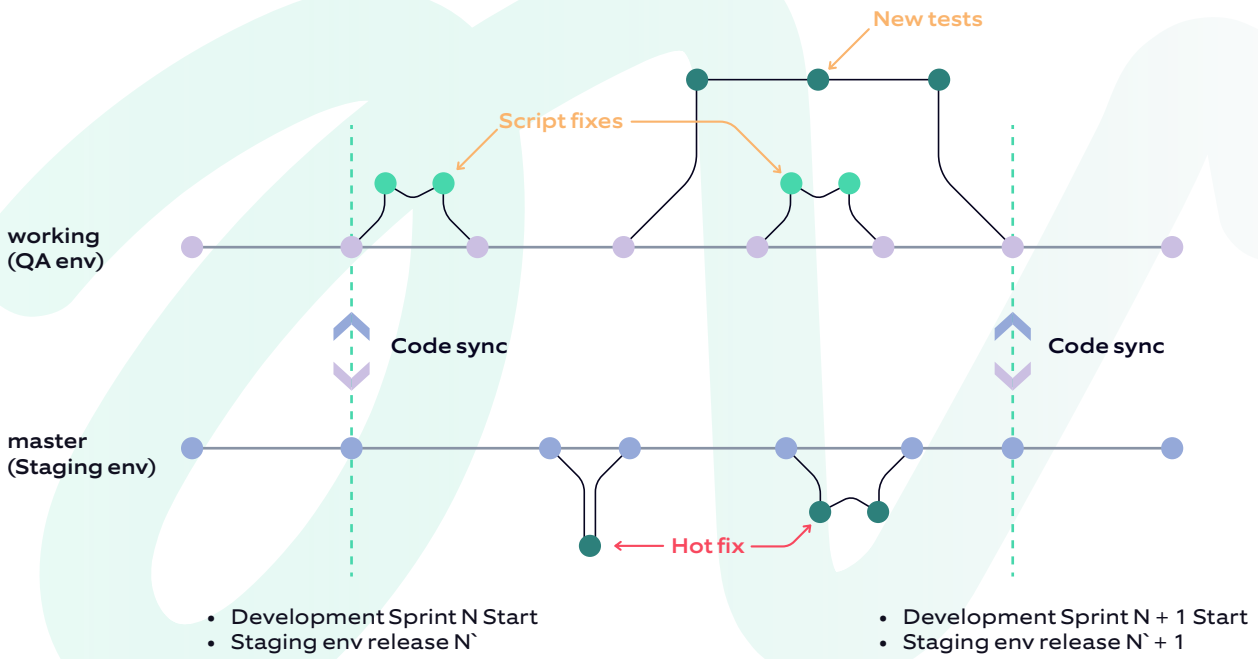
1. Agreeing to add data-testid to web elements for stability in automation
2. Requesting development teams to publish additional resources for automation testing (Example: Special web pages for testing of CMS functionality)
3. Defining clear conditions and schedules for automation testing

4. Establishing rules and guidelines for the automation team



Additional process considerations include:

- Maintaining 1:1 mapping between test cases stored in test case management tool and automation test scenarios/outlines for easier test execution results updating and automation coverage presentation
- Using backend services/APIs for setup and teardown steps to improve efficiency and stability
- Introducing dependency and package management for better version control and code reusability
- Using Docker images for cost-effective and scalable remote test execution infrastructure
- Controlling system configuration using APIs (example: Turn off LaunchDarkly feature flags to disable reCAPTCHA in test environments) wherever possible and manually so there are no surprises during test automation execution
- Implementing a strong branching strategy corresponding to the work being done against different environments and doing the code synchronization timely aligned with the application changes in each environment



7. Accelerators

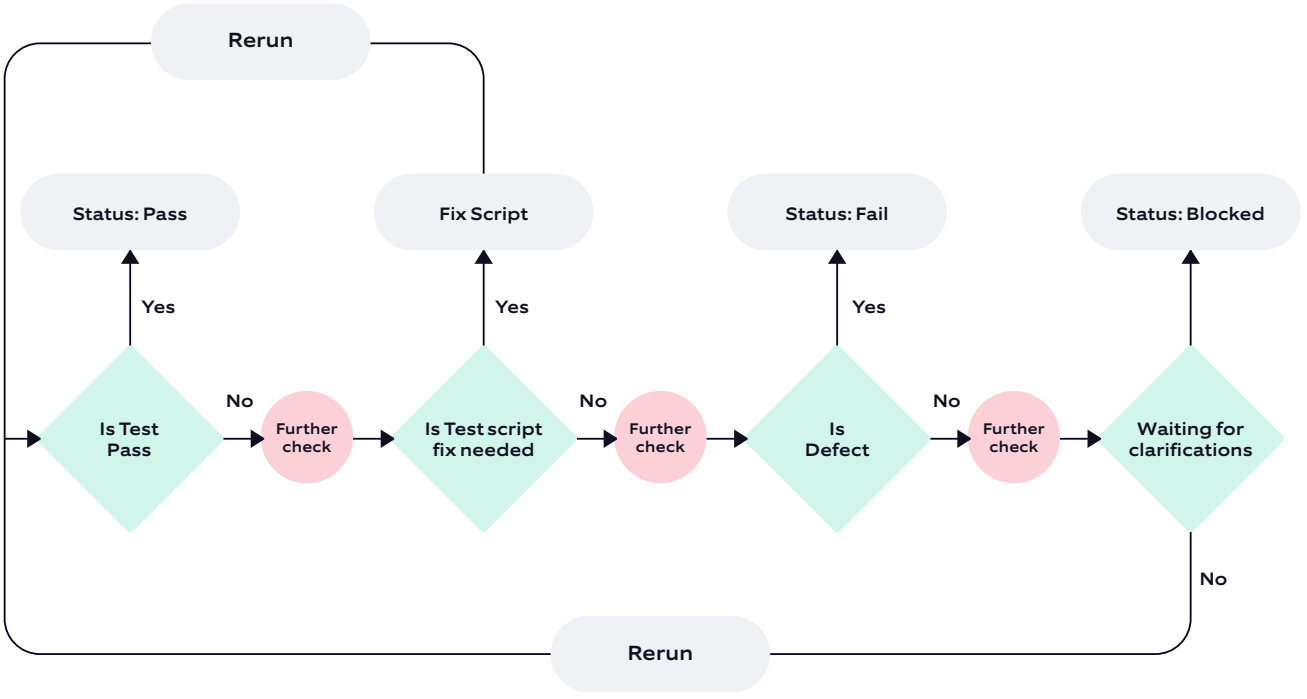
There are many activities performed during the test automation process without the assistance of software programs. These activities can be repetitive and considering the project's duration, they could be happening too many times consuming a lot of accumulated efforts. Replacing these activities with test automation accelerators saves a considerable amount of effort for the team and improves accuracy and efficiency in delivery.

7.1 Re-running of Tests

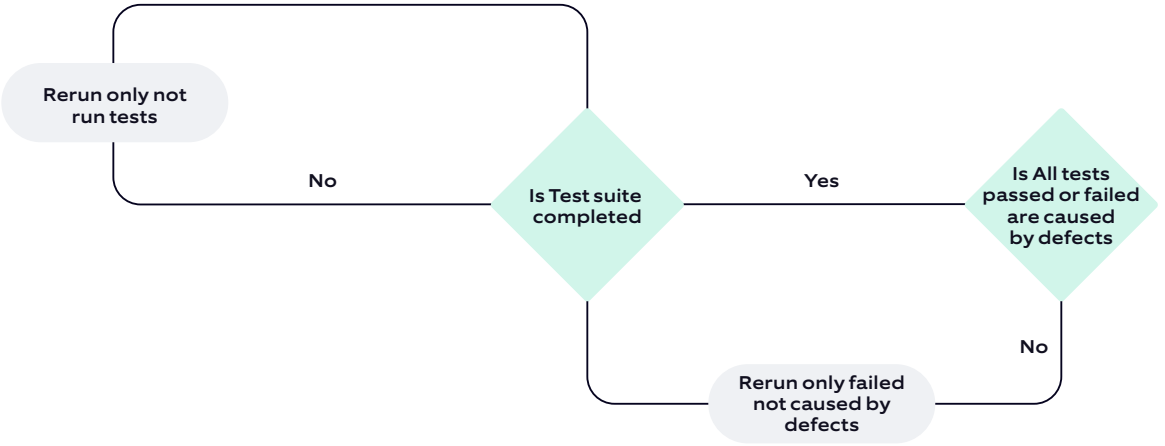
Many activities in the test automation process are performed manually without software assistance. These repetitive tasks can consume significant accumulated effort over the project duration. Implementing test automation accelerators for these activities can save considerable effort and improve accuracy and efficiency in delivery.

Frontend tests, especially long-running end-to-end tests, are inherently unstable and flaky. They frequently break due to false positives, often caused by data-related issues, environmental problems, or timeouts due to slowness. In this case, in order to evaluate the complete test result, running all tests over and over until 0 false positives, is not cost-effective. Therefore, an efficient test rerunning mechanism that only reruns failing tests is necessary.

However, it's not logical to rerun tests that fail due to application defects or temporary blockers. Further optimization is required by introducing necessary filters for rerunning failed tests. Additionally, there's a possibility that test execution could be abruptly terminated before completion. To address this, it's important to execute the remaining 'not run' tests first before retrying the failed tests, ensuring a fair representation of the overall result from the first round of test suite execution.



To implement this logic, Jenkins rerun pipelines are designed corresponding to each original Jenkins test execution pipeline. These can be retriggered based on build status recursively or on demand, significantly reducing the time required for test results analysis. Each test execution maintains its own artifacts and updates accumulated results into dashboards.



Zephyr test results serve as the single source of truth for test status, enabling accurate decisions on whether to run a certain test during the rerun phase, regardless of who is executing it (locally or remotely via Jenkins). This approach facilitates better work distribution during test results analysis and prevents duplication when multiple team members work on analyzing the same test suite results.

7.2 Error classification report

Standard reports from automation tools provide individual test failure information, but this often requires the automation team to manually review each test case to identify failure reasons. To streamline this process and improve efficiency, an error classification system can be implemented:

- Classifying errors based on their nature provides the automation team with an additional resource to speed up results analysis.
- This classification helps the team prioritize fixes based on impact and make quicker decisions.
- Implementing error classification can be achieved by throwing framework-specific exceptions/errors based on applied filters to results.

By implementing this error classification system, teams can more efficiently analyze test results, prioritize issues, and make informed decisions about necessary fixes. This approach can significantly reduce the time spent on manual review of individual test failures and allow for more strategic allocation of resources in addressing critical issues.

7.3 Dashboards

In addition to the standard reports available via the primary test automation tool and JIRA/Zephyr, there are other statistics that help make accurate and faster decisions. Implementing dashboards that render these statistics in an easily accessible manner is key to success. A locally hosted dashboard via terminal commands is utilized, with updated snapshots of the dashboard being published to confluence regularly so other stakeholders can make use of them.

Key statistics

1. Results trend over builds/executions: Showing test results per every test case over past builds allows the team to identify the brittle tests and prioritize them for fixing.
2. Results comparison testcase vs brand-region-country-language-device: Presenting current test execution related results on test cases against all application instances the test was executed allows the team to determine failure causes quickly.
3. Test execution status: Current test execution status (number of tests executed, passed, failed, skipped, blocked, impacted by defects and remaining to be attended, defects and their statuses along with impacted number of tests, team assignments) for all test suites being run currently, provide a complete overview in one view.
4. Test execution time: Presenting test execution time per every test case over past builds allows the team to identify the inefficient tests and prioritize them for fixing.
5. Test automation coverage: Presenting number of total tests, total automated and total deprioritized for all the applications in a bar chart provides an overview of test automation coverage.

7.4 Self-healing and prevention

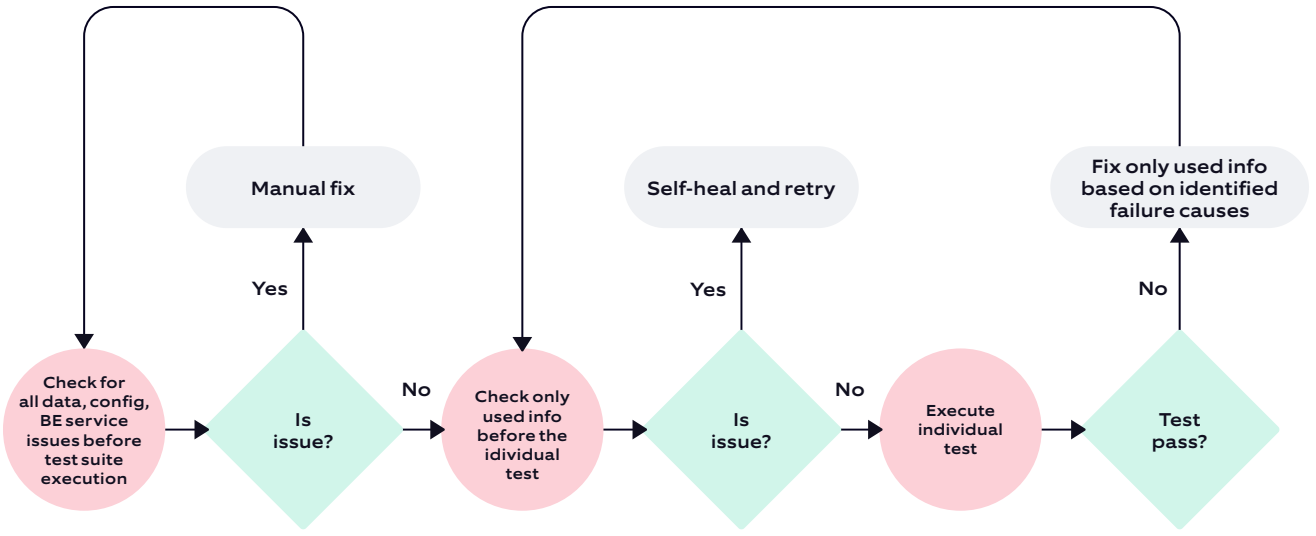
Automated test executions often fail due to false positives, leading to significant effort spent on investigating misleading information without producing any definitive results. Self-healing and preventive measures must be implemented to avoid false positive results.

False positives can occur due to:

- Environment issues: Inconsistent configurations, insufficient resources, or issues with dependencies can cause tests to fail even when the software is functioning correctly.
- Data issues: Using incorrect, outdated, or inconsistent test data can lead to unexpected failures, even if the application is working as intended.
- Timing and synchronization problems: Tests may check for conditions before the application has fully completed an operation, resulting in false failures.
- Misconfigured test setups: Incorrect settings, such as wrong URLs, paths, or parameters, can cause tests to behave unexpectedly and produce false positive results.

Actions taken to avoid false positives include:

1. Introducing a scanner that automatically scans products, users, promotions, stores, and payment data, checking for validity before the test suite execution. This allows the automation maintenance team to make necessary manual changes in a timely manner.
2. Adding automatic data and configuration validation as preconditions to test steps, enabling them to self-heal any data and configuration issues that may arise between the last manual scan prior to test suite execution start and the execution of individual tests.
3. Implementing a heartbeat check before tests to ensure that the application and relevant backend services are up and running. Backend services are used to restore configurations if they are not in the expected condition before test execution.
4. Adding a failover strategy where test scripts are retried automatically after restoring data and configurations, if failures are identified as due to known data or configuration issues (Error classification). This helps to self-heal tests that fail due to changes that occur during test execution, after the initial automatic check.



7.5 Test case vs Test script sync

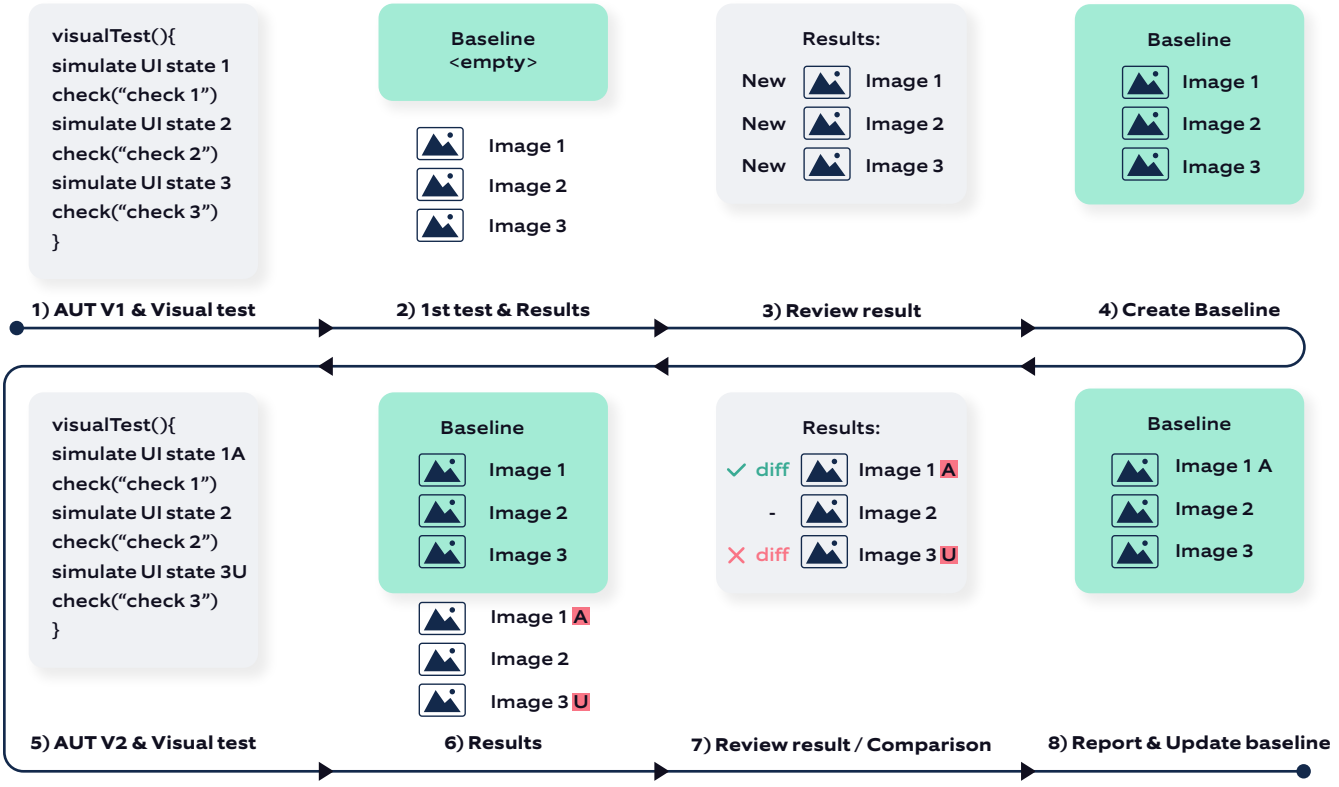
Maintaining and keeping up-to-date functional test cases and corresponding test scripts is necessary. Whenever changes are introduced to the existing features, functional test cases are changed accordingly and there should be an easier cost-effective way to be able to track down where changes are required at the test script level.

In this instance, there are frequent changes being done to the functional testcases due to various reasons (example: feature applicable devices being updated, feature change impacts across all applications, certain feature change only happens for a certain brand region only) and these changes could be done by any stakeholder who is not responsible for test script maintenance.

Introducing a test automation framework feature that can listen and notify on the test case level changes addresses the problem. Zephyr API is used to extract test case information changes since the previous sync.

8. Visual regression

Implementing visual regression testing is crucial to ensure UI consistency across different versions of the application. This approach helps maintain a consistent user experience and catch unintended visual changes early in the development process.



- As the visual regression maintenance cost is high compared to the expected return, compile a small suite of tests that navigate through key pages of the application and use it for daily visual regression testing using Applitools.
- Run tests in a much more stable test environment daily, with baselining done every day after manual results analysis.
- Maintain data/config consistency and exclude dynamic content from comparison by using the Applitools API, focusing the tests on the UI aspect of the application only.

By implementing these visual regression testing strategies, the team can quickly identify and address any unintended visual changes, ensuring a consistent and high-quality user interface across all versions of the e-commerce applications.

9. Collaboration and communication

Effective collaboration and communication are essential components of a successful test automation strategy for multiple omnichannel e-commerce applications. To ensure alignment on test automation goals and strategies, it's crucial to foster collaboration between development, QA, and operations teams. This collaborative approach helps create a unified vision and ensures that all stakeholders are working towards the same objectives.

One key aspect of promoting collaboration is to increase awareness among other stakeholders about the test automation approach and its associated challenges. This can be achieved through regular demo sessions, which provide an opportunity to showcase the automation process, highlight successes, and discuss any obstacles encountered. These sessions help build understanding and appreciation for the complexities of test automation across the organization.

Maintaining transparency and driving continuous improvement requires regular communication of test results, progress, and challenges to all relevant stakeholders. This open line of communication ensures that everyone is kept informed about the status of the automation efforts and can contribute to problem-solving and process refinement as needed.

To support these collaborative efforts, it's important to make comprehensive documentation readily available. This documentation should cover various aspects of the test automation process, including the overall strategy, infrastructure details, user manuals, build reports, team structure, contact information, and links to relevant resources. By providing easy access to this information, teams can work more efficiently and effectively, with a clear understanding of the automation landscape.

Conclusion: Mastering omnichannel e-commerce test automation

Automating tests for omnichannel e-commerce applications serving multiple brands and regions with internationalization support is a challenging but essential task. By understanding the unique challenges, adopting a structured test approach, and implementing a comprehensive test automation strategy, businesses can ensure high-quality user experiences across all channels and regions.

Effective test automation not only improves efficiency and coverage but also helps in delivering consistent and reliable e-commerce platforms that meet the diverse needs of global customers.

Glossary of Abbreviations

EC2 – Elastic Compute Cloud (Amazon Web Services)
API – Application Programming Interface
SaaS – Software as a Service
POC – Proof of Concept
ROI – Return on Investment
AUT – Application Under Test
REST – Representational State Transfer
UI/UX – User Interface/User Experience
QA – Quality Assurance
XML – Extensible Markup Language
CSV – Comma-Separated Values
SLA – Service Level Agreement
i18n – Internationalization
l10n – Localization
CI/CD – Continuous Integration and Continuous Deployment
API – Application Programming Interface
CMS – Content Management System
CRM – Customer Relationship Management
BOPIS – Buy Online, Pick Up In-Store
UI – User Interface
UX – User Experience
POM – Page Object Model
BDD – Behavior-Driven Development

About the author

Bimal Chathurinda Tissakuttige is a seasoned IT professional with over 17 years of experience, specializing in Test Automation, Performance Testing, Functional Testing, and Software Development. Currently a Principal Engineer at Nagarro iQuest Schweiz AG in Zurich, Bimal directs a team focused on developing and implementing innovative test automation solutions. His career spans across various industry domains, including e-commerce and banking, demonstrating a strong track record in optimizing testing processes and managing large teams. Bimal holds a B.Sc. (Hons.) in Computing from Staffordshire University, UK.

https://www.linkedin.com/in/bimaltissakuttige/