electronics

MDPI

Article

A Survey of the Selenium Ecosystem

Boni García [1,*], Micael Gallego [2], Francisco Gortázar [2] and Mario Munoz-Organero [1]

[1] Department of Telematic Engineering, Universidad Carlos III de Madrid, Avenida de la Universidad 30, 28911 Leganés, Spain; munozm@it.uc3m.es

[2] Department of Computer Science, Computer Architecture, Computer Languages & Information Systems, Statistics & Operational Research, Universidad Rey Juan Carlos, Calle Tulipán S/N, 28933 Móstoles, Spain; micael.gallego@urjc.es (M.G.); francisco.gortazar@urjc.es (F.G.)

* Correspondence: boni.garcia@uc3m.es

Received: 3 June 2020; Accepted: 25 June 2020; Published: 30 June 2020

Abstract: Selenium is often considered the de-facto standard framework for end-to-end web testing nowadays. It allows practitioners to drive web browsers (such as Chrome, Firefox, Edge, or Opera) in an automated fashion using different language bindings (such as Java, Python, or JavaScript, among others). The term ecosystem, referring to the open-source software domain, includes various components, tools, and other interrelated elements sharing the same technological background. This article presents a descriptive survey aimed to understand how the community uses Selenium and its ecosystem. This survey is structured in seven categories: Selenium foundations, test development, system under test, test infrastructure, other frameworks, community, and personal experience. In light of the current state of Selenium, we analyze future challenges and opportunities around it.

Keywords: automated software testing; web; selenium; software ecosystems

1. Introduction

Selenium (https://www.selenium.dev/) is an open-source framework mainly used for testing web applications. It enables the impersonation of users interacting with browsers such as Chrome, Firefox, Edge, or Opera in an automated manner. Nowadays, it is considered the de facto framework to develop end-to-end tests for web applications and supports a multi-million dollar industry [1]. A recent study about software testing by Cerioli et al. identifies Selenium as the most valuable testing framework nowadays, followed by JUnit and Cucumber [2].

Software ecosystems are collections of actors interacting with a shared market underpinned by a common technological background [3]. Specifically, the Selenium ecosystem, target of the study, comprises the Selenium framework and other projects, libraries, and other actors. Selenium is a relevant testing framework due to the valuable contribution of its root projects (WebDriver, Grid, IDE), and for the contributions of a significant number of related projects and initiatives around them. This article presents the results of a descriptive survey carried out to analyze the status quo of Selenium and its ecosystem. A total of 72 participants from 24 countries completed this survey in 2019. The results reveal how practitioners use Selenium in terms of main features, test development, System Under Test (SUT), test infrastructure, testing frameworks, community, and personal experience. In light of the results, we provide some hints about future challenges around Selenium.

The remainder of this paper is structured as follows. Section 2 provides a comprehensive review of the main features of the Selenium family (WebDriver, Grid, and IDE) and the related work. Section 3 presents the design of the descriptive survey aimed to analyze the Selenium ecosystem. Section 4 presents the results of this survey, and then Section 5 provides a discussion and validity analysis of these results. Finally, Section 6 summarizes some conclusions and possible future directions for this piece of research.

2. Background

Selenium was first released as open-source by Jason Huggins and Paul Hammant in 2004 while working in ThoughtWorks. They chose the name "Selenium" as a counterpoint to an existing testing framework developed by Hewlett-Packard called Mercury. In the opinion of Huggins, Selenium should be the testing framework that overcomes the limitations of Mercury. The idea behind the name is that "Selenium is a key mineral which protects the body from Mercury toxicity".

The initial version of Selenium (now known as **Selenium Core**) allowed to interact with web applications impersonating users by opening URLs, following links (by simulating clicks on them), enter text on forms, and so forth, in an automated fashion. Selenium Core is a JavaScript library that interprets operations called Selenese commands. These commands are encoded as an HTML table composed by three parts:

1. Command: an action done in the browser. For example, opening a URL or clicking on a link.
2. Target: the locator which identifies the HTML element referred by the command. For example, the attribute id of a given element.
3. Value: data required by the command. For example, the text entered in a web form.

Huggins and Hammant introduced a scripting layer to Selenium Core, creating a new Selenium project called **Remote Control (RC)**. As depicted in Figure 1a, Selenium RC follows a client-server architecture. Selenium scripts using a binding language (such as Java, Perl, Python, Ruby, C#, or PHP) act as clients, sending Selenese commands to an intermediate HTTP proxy called Selenium RC server. This server launches web browsers on demand, injecting the Selenium Core JavaScript library in the SUT to carry out the automation through Selenese commands. To avoid same-origin policy issues (for example, when testing a public website), the Selenium RC Server masks the SUT under the same local URL of the injected Selenium Core library. This approach allows practitioners to create automated end-to-end tests for web applications using different programming languages.

Although Selenium RC was a game-changer in the browser automation space, it had significant limitations. Since JavaScript was the base technology to support automated web interactions, different actions could not be performed, for example, file upload and download, or pop-ups and dialogs handling, to name a few. Moreover, the Selenium RC Server introduces a considerable overhead which impacts in the final performance of the test. For these reasons, Simon Stewart at ThoughtWorks developed a new tool called **Selenium WebDriver**, first released in 2008. From a functional point of view, WebDriver is equivalent to RC, that is, it allows the control of browsers from scripts using different language bindings [4]. Nevertheless, its architecture is entirely different. To overcome the limitation of automating through JavaScript, WebDriver drives browsers using their native automation support. To that aim, a new component between the language binding and the browser needs to be introduced. This component is called the driver. The driver is a platform-dependent binary file that implements the native support for automating a web browser. For instance, this driver is named *chromedriver* in Chrome or *geckodriver* in Firefox (see Figure 1b). The communication between the Selenium client and the driver is done with JSON messages over HTTP using the so-called JSON Wire Protocol. This mechanism, proposed initially by the Selenium team, has been standardized in the W3C WebDriver recommendation [5]. Nowadays, Selenium WebDriver is also known as "Selenium 2", while Selenium RC and Core are known as "Selenium 1", and its use is discouraged in favor of WebDriver.

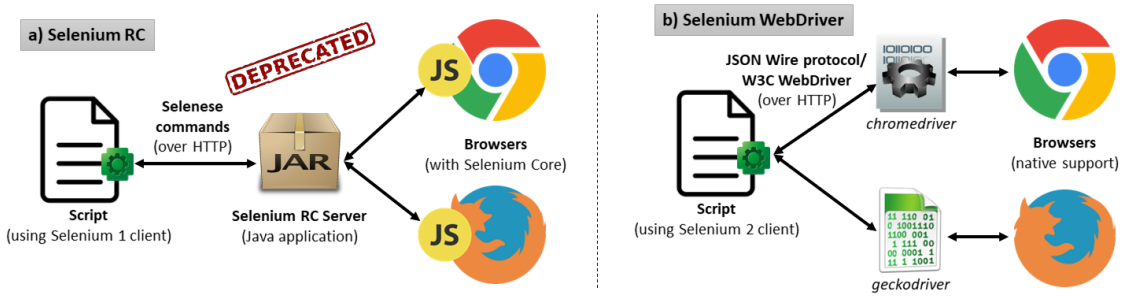

Figure 1. Selenium Remote Control (RC) and Selenium WebDriver architecture.

Another relevant project of the Selenium family is **Selenium Grid**. Philippe Hanrigou developed this project in 2008. We can see it as an extension of Selenium WebDriver since it allows testers to drive web browsers hosted on remote machines in parallel. As illustrated in Figure 2, there may be several nodes (each running on different operating systems and with different browsers) in the Selenium Grid architecture. The Selenium Hub is a server that keeps track of the nodes and proxies requests to them from Selenium scripts.

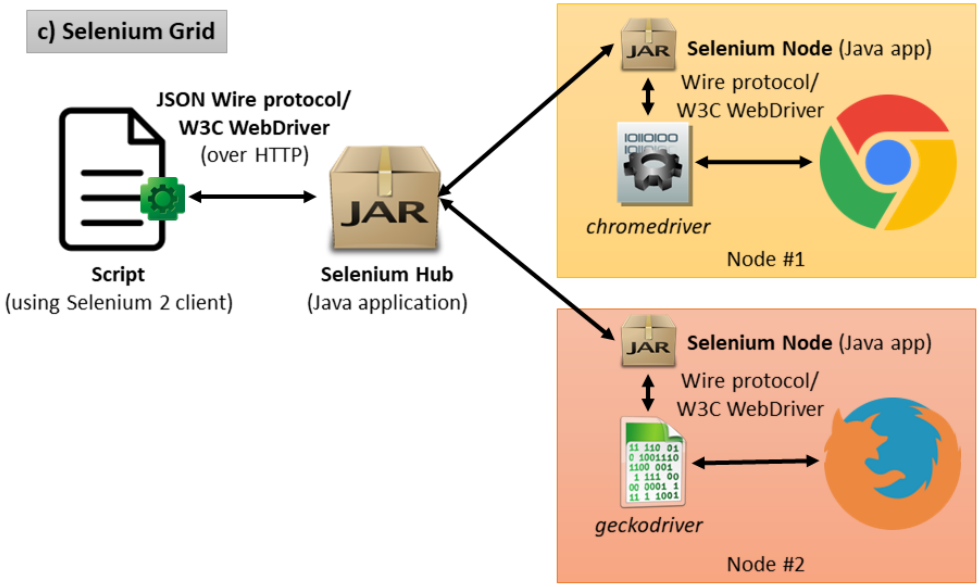

Figure 2. Selenium Grid architecture.

The last project of the Selenium portfolio is called the **Selenium IDE**. In 2006, Shinya Kasatani wrapped the Selenium Core into a Firefox plugin called Selenium IDE. Selenium IDE provides the ability to record, edit, and playback Selenium scripts. As of version 55 of Firefox (released on 2017), the original Firefox plugin mechanism implemented in Firefox (based on XPI modules) migrated to the W3C Browser Extension specification [6]. As a result, the Selenium IDE stopped working. To solve this problem, the Selenium team rewrote the IDE component as Chrome and Firefox extensions using the Browser Extensions API [6]. Moreover, at the time of this writing, Selenium IDE is being ported to Electron. Electron is an open-source framework that allows the development of desktop applications based on Chromium and Node.js [7].

2.1. Related Work

This paper belongs to the test automation space. In this arena, we can find a relevant number of existing contributions in the literature. For instance, Rafi et al. present a systematic literature review about the benefits and limitations of automated software testing [8]. According to this work,

the main advantages of test automation are reusability, repeatability, and effort saved in test executions. As counterparts, test automation requires investing high costs to implement and maintain test cases. Besides, staff training is also perceived as problematic. Other contributions report the status of test automation following different approaches. For example, Kasurinen et al. present an industrial study aimed to identify the factors that affect the state of test automation in various types of organizations [9]. In this work, test automation is identified as a critical aspect to secure product features. However, high implementation and maintenance costs are recognized as limiting issues. The limited availability of test automation tools and test infrastructure costs causes additional problems. These results are aligned with the contribution of Bures and Miroslav [10]. This work also identifies the maintenance of test scripts as a limiting point for the efficiency of test automation processes.

Focusing on the specific domain of Selenium, we find a large number of contributions in the literature. For example, the SmartDriver project is an extension of Selenium WebDriver based on the separation of different aspects of the test automation in separate concerns: technical elements related to the user interface and test logic and business aspects associated with the application under test [11]. Leotta et al. propose Pesto, a tool for migrating Selenium WebDriver test suites towards Sikuli [12]. Selenium WebDriver is sometimes referred to as a second-generation automation framework for web applications since the location strategies are based on the DOM (Document Object Model) of web pages. On the other side, Sikuli is a third-generation tool since it uses image recognition techniques to identify web elements [13]. This technique can be convenient for those cases where the DOM-based approach is not possible (e.g., Google Maps automated testing). Another tool of the Selenium ecosystem is Selenium-Jupiter, an JUnit 5 extension for Selenium providing seamless integration with Docker. References [14,15] show how to use Selenium-Jupiter to implement automated tests aimed to evaluate the Quality of Experience (QoE) of real-time communication using web browsers (WebRTC) [16].

Maintainability problems are also reported in the Selenium literature. The modification and evolution of a web application might break existing Selenium tests. For instance, this could happen when the layout of a web page changes, and Selenium tests coupled to the original layout remain unaltered. When this occurs, repairing the broken tests is usually a time-consuming manual task. In this arena, Christophe et al. present a study about the prevalence of Selenium tests. This work tries to understand the maintenance effort of existing Selenium tests as long as the SUT evolves. Authors claim to identify the elements of a test that are most prone to change: constants (frequently in locator expressions to retrieve web elements) and assertions (statements to check the expected outcomes) [17]. Bures et al. propose a tool called TestOptimizer aimed to identify potentially reusable objects in automated test scripts [18]. This tool carries static analysis in test scripts (such as Selenium WebDriver tests) to seek reusable subroutines. In this line, Stocco et al. investigate the benefits of the page objects pattern to improve the maintainability of Selenium tests with Apogen, a tool for the automatic generation of these objects [19].

In other related publications, we find contributions comparing Selenium with other alternatives. For instance, Kuutila presented a literature review on Selenium and Watir, identifying differences in the performance using different configurations in these testing tools [20]. We find another example in Reference [21], in which Kaur and Gupta provides a comparative study of Selenium, Quick Test Professional (QTP), and TestComplete in terms of usability and effectiveness.

Finally, we find the most straightforward reference to the target of this paper, that is, the Selenium ecosystem, in the official documentation (https://www.selenium.dev/ecosystem/). This information captures existing tools which use Selenium WebDriver as its main component, namely:

- Browser drivers: such as *chromedriver* for Chrome, or *geckodriver* for Firefox.
- Additional language bindings: Go, Haskell, PHP, R, Dart, among others.
- Frameworks: WebDriverIO, Capybara, Selenide, FluentLenium, among others.

To the best of our knowledge, there are no further sources that analyze the Selenium ecosystem as a whole. We interpret this fact as an indicator of the need for the present article.

3. Survey Design

The way to conduct an investigation is often known as a research approach [22]. Glass et al. identified three types of research approaches in the computing field (composed by Computer Science, Software Engineering, and Information Systems), namely Reference [23]:

- Descriptive: aimed to explain the characteristics of a phenomenon under study.
- Evaluative: aimed to assess some effort in a deductive, interpretative, or critical manner.
- Formulative: aimed to propose a solution to a problem.

We use a descriptive research approach to get the current snapshot of Selenium and its ecosystem. To that aim, we use an online questionnaire implemented with Google Forms to gather information from the Selenium community. We launched the survey just before the beginning of SeleniumConf Tokyo (https://conf.selenium.jp/) in April 2019. At that time, the stable version was Selenium 3, and during that conference, the first alpha release of Selenium 4 was made public.

We employed two different ways to recruit participants for the survey. First, we used the general channel of the SeleniumConf Slack workspace (https://seleniumconf.slack.com/). Participants in this channel are former or future participants in SeleniumConf. Therefore, the potential respondents coming from this workspace are likely to be experts (or at least quite interested) in Selenium. The left part of Figure 3 shows the message posted on this Slack channel. To reach a wider audience, we also ask for participation in the survey in the Selenium Reddit community (https://www.reddit.com/r/selenium/). People participating in this community show a proactive interest in Selenium, and therefore, their opinion can also be valuable for the survey. The right part of Figure 3 shows the message posted on the Selenium Reddit community.

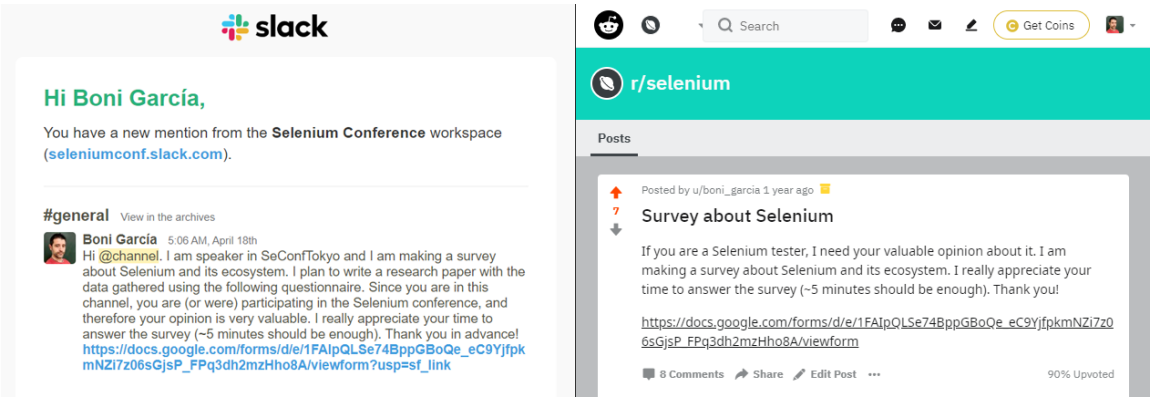

Figure 3. Screenshots of the messages posted on the SeleniumConf Slack workspace (**left**) and Selenium Reddit community (**right**) to recruit participants for the survey

As illustrated in Figure 4, the structure of the survey contains seven different sections. We labeled the first section as "Foundations" in this diagram, and it is about the core features of Selenium (i.e., essential aspects of Selenium) used by the community. This section contains two questions. First, we ask the participants to chose which project of the official Selenium family (i.e., WebDriver, Grid, IDE) use. This question is a multiple-choice close-ended question [24]. In other words, respondents can choose one or more answers to this question. The second question in this section is about the language bindings, that is, the programming language in which Selenium scripts are developed. The possible answers to this question are the officially supported languages for Selenium, that is, Ruby, Java, JavaScript, Python, C#, or JavaScript (https://www.selenium.dev/downloads/), and also other language bindings supported by the community, such as PHP or Dart among others. Even though developers can use different language bindings, this question is designed to be a single-choice close-ended. In other words, participants can choose one and only one of the answers. This constraint aims to get a reference of the preferred language and use this number to calculate usage quotas of additional features of Selenium relative to the language binding.

As depicted in Figure 4, the rest of the questions available from sections two to six, are of the same type: multiple-choice close-ended questions (including an open-ended field for other options). These questions provide a list of possible answers (which can be selected or not by respondents) and an additional field for alternatives. This format aims to simplify the survey filling by participants while providing additional possibilities (the "other" field) to complement the answers with custom choices.

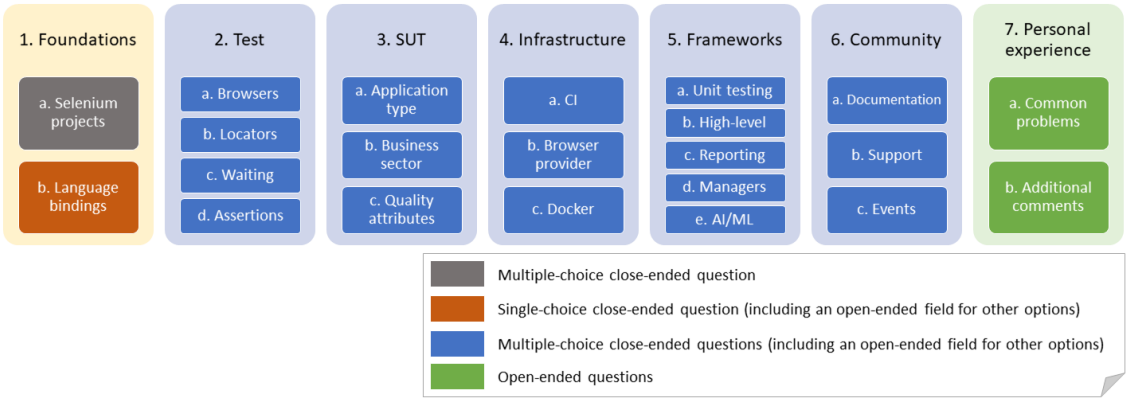

Figure 4. Survey categories.

The second section, labeled as "Test" in Figure 4, is about the design and implementation of the end-to-end tests developed with Selenium. This section aims to discover how participants develop Selenium tests by analyzing the following aspects:

- Browsers selected to be driven automatically through Selenium, such as Chrome, Firefox, Opera, Edge, among others.
- Locator strategy. A locator is an abstraction used in the Selenium API to identify web elements in Selenium. There are different strategies available in the Selenium API: by attribute id, by attribute name, by link text, by partial link text, by a given tag name, by the attribute class, by a CSS (Cascading Style Sheets) selector, or by XPath [25]. In addition to these methods, there is a complementary technique to locate web elements called the Page Object Model (POM). POM is a design pattern in which developers can model web pages using an object-oriented class to ease test maintenance and reduce code duplication [26]. These page objects internally also use the aforementioned DOM-based locators (by id, name, and so for). We specifically ask if users use POM because it is the best practice to help make more robust tests.
- Waiting strategy. The load time of the different elements on a web page might be variable. To handle this dynamic interaction, Selenium implements different waiting strategies. There are two main types of waits in Selenium. On the one hand, the so-called implicit waits tell Selenium WebDriver to wait a certain amount of time before raising an exception. On the other hand, explicit waits wait a given timeout to a given expected condition. Moreover, there is a type of explicit condition called fluent waits, which allows setting additional parameters such as the polling time and the exception raised [27].
- Assertion strategy. Tests are supposed to exercise a SUT to verify whether or not a given feature is as expected, and so, an essential part of tests are the assertions. Assertions are predicates in which the expected results (test oracle) are compared to the actual outcome from the SUT [28]. The strategy for this assessment can be heterogeneous in Selenium, and for that reason, this part of the survey investigates this relevant point.

The next part of the survey is labeled as "SUT" in Figure 4. This part first surveys the nature of the application tested with Selenium. Although the answer to this question seems pretty obvious since Selenium is a test framework for web applications, we ask the participants this question to evaluate if other application types are also assessed (for instance, mobile apps, when also using Appium). In the

next question, we ask about the business sector in which the participants catalog their SUT, such as enterprise applications, e-commerce, or gaming, to name a few. Moreover, we ask the participants to inform about the quality attributes verified with Selenium, for example, functionality, performance, compatibility, usability, accessibility, or other non-functional properties [29].

The fourth group of questions of the survey concerns to test infrastructure and contains three items. This first question asks the participants for the Continous Integration (CI) server used to execute Selenium tests. The second question surveys about the browser provider, that is, the source of the browsers. Possible answers to this question include the use of local browsers (installed in the machine running the test), remote browsers (accessed using Selenium Grid), browsers in Docker containers, or use of services that provide on-demand browsers, such as SauceLabs or BrowserStack, among others. To conclude this section, we included an additional question regarding the Docker tools used with Selenium.

The fifth part, "Frameworks", is about complementary tools to Selenium. This part explores existing frameworks, libraries, and other utilities used in conjunction with Selenium, grouped into different categories:

- Unit testing frameworks. Although Selenium can be used as a standalone framework (for example, when making web scrapping to gather data from websites [30]), it is a common practice to embed a Selenium script inside a unit test. This way, the execution of the Selenium logic results in a test verdict (pass or fail). Examples of typical unit testing frameworks are JUnit, TestNG, or Jasmine, among others.
- High-level testing frameworks. Selenium is sometimes used in conjunction with additional frameworks that wrap, enhance, or complement its built-in features. Examples of these frameworks are, for example, Cucumber, Protractor, or WebDriverIO, among others.
- Reporting tools. Test reports are pieces of information in which the results of a test suite are documented. These reports can be critical to try to find the underlying cause of some failure. This part explores the mechanisms used to report Selenium tests by practitioners.
- Driver managers. As explained in Section 2, Selenium WebDriver requires a platform-dependent binary file called driver (e.g., *chromedriver* when using Chrome or *geckodriver* when using Firefox) to drive the browser. We find different tools to automate the management of these drivers in the Selenium ecosystem, the so-called driver managers. This question investigates the actual usage of these tools in the community.
- Framework based on Artificial Intelligence (AI) or Machine Learning (ML) approaches. AI/ML techniques promise to revolutionize the testing practice in the next years [31]. We include a specific question about it to estimate the usage of these techniques in the Selenium space.

The sixth section of the survey comprises aspects related to the Selenium community. The first question is about the preferred sources of information to learn and keep updated about Selenium (i.e., documentation). The second question surveys about the most convenient ways to obtain support about Selenium from the community. The last item of this section asks the participant to report relevant events related to Selenium, such as SeleniumConf or the Selenium Groups.

The last part of the survey contains open-ended questions about the participants' experience with Selenium. We ask them to use their own words (in English) using a text area to answer these questions. First, we ask about the common problems related to the development and operation with Selenium. This question aims to discover the main issues that Selenium testers face, such as maintainability or flakiness, among other potential problems. To conclude, this survey contains another open-ended question to report personal opinions or further comments about the Selenium ecosystem.

4. Results

A total of 72 people from 24 countries worldwide completed the questionnaire. As shown in the left chart of Figure 5, the ordered list of participants' countries is the following: United States,

India, Spain, United Kingdom, Canada, Japan, Argentina, Brazil, Germany, Australia, Belarus, Finland, Hungary, Ireland, Korea, Mexico, Norway, Pakistan, Philippines, Poland, Romania, Russia, Slovenia, and Sweden. From a demographic perspective, the participants' ages are distributed from 21 to 53 years old, being the thirties the more numerous group, which accounts for the 52.94% of the total participants. Regarding gender, the respondents were 87.5% men, 9.72% women, and 2.78% chose the option "other/prefer not to say".

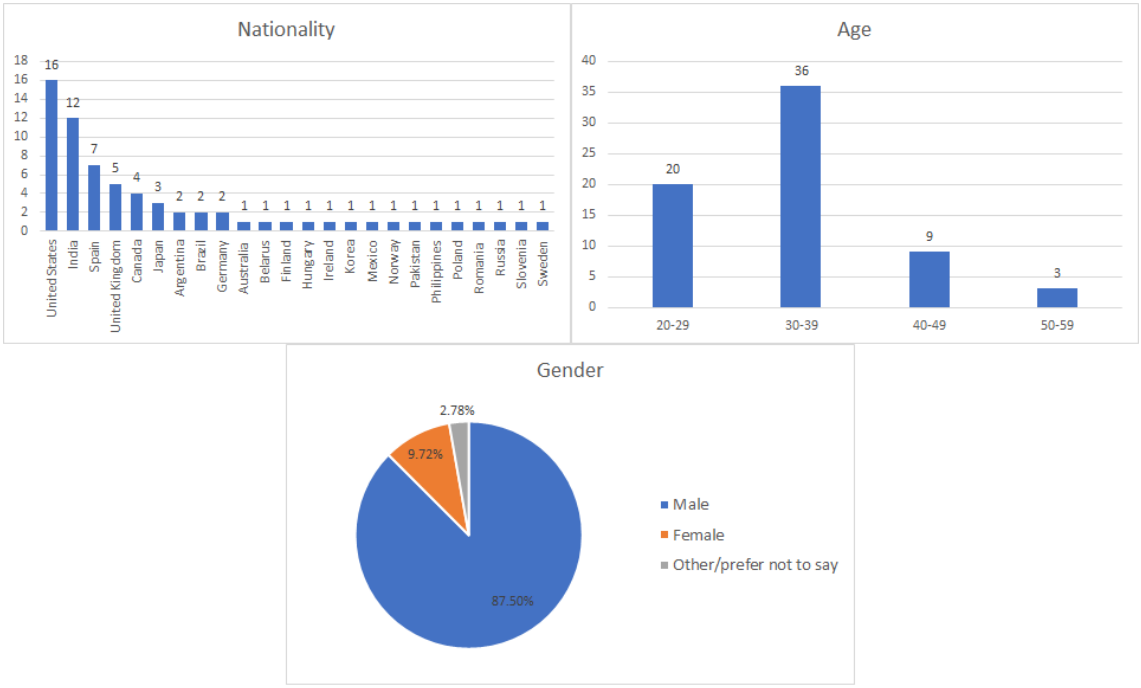

Figure 5. Figures showing the total number of participants per nationality, age, and gender.

The rest of this section presents the results of the seven parts for the survey. Moreover, we report the raw data results together with the survey questions in Appendix A.

4.1. Foundations

Figure 6 presents the resulting charts for the first section of the survey. Regarding the use of Selenium projects, we check that clearly, Selenium WebDriver is the most utilized project, since 71 out of the total participants, that is, 72, declared to use Selenium WebDriver. This number implies a usage quota of 98.61% for Selenium WebDriver. Additionally, 27 respondents (37.5% of the total) declared to use Selenium Grid as well. Finally, we find that Selenium IDE was selected 11 times in the survey (15.28%). Regarding the preferred programming language used in Selenium scripts, Java is the preferred by the 50% of the participants. Then we find Python (18.06%), JavaScript (13.89%), C# (12.5%), Ruby (2.78%), and finally Groovy and PHP (1.39% each one).

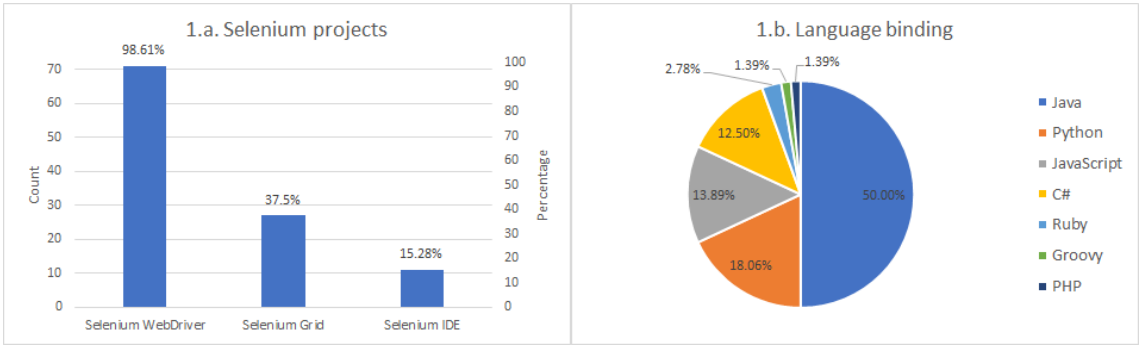

Figure 6. Selenium projects and language bindings.

4.2. Test Development

In the second section of the survey, we investigate several fine-grained details of the test design and implementation. Regarding browser usage, as illustrated in Figure 7, we check that Chrome is the most used since it was selected by 68 out of the 72 participants. This number is equivalent to a very relevant usage quota of 94.44%. The second browser is Firefox, selected 50 times in the questionnaire, that is, 69.44%. The third browser in terms of Selenium automation is Internet Explorer. It was chosen by 25 participants, which is equivalent to a rate of 34.72% of the participants). The rest of the browsers are Edge (11 usages, 15.28%), Safari (8 usages, 11.11%), and finally, Opera, HtmlUnit, and PhantomJS (these three with a small quota of 2.78%). The right chart of Figure 7 illustrates the results of the locator strategy. The top strategies are by id, name, XPath, and CSS (chosen by a total percentage of 56.94%, 40.28%, 36.11%, and 29.17% of the respondents, respectively). It is worth noting that only 12.5% of respondents indicated they are using the POM pattern, which is a recommended practice to reduce maintenance costs and build more robust tests. Similarly, more than 36% of respondents use XPath expressions as locators, which, when not used properly, is considered a bad practice that tends to make tests unreliable [32]. The bottom chart of Figure 7 shows the results of waiting strategies. We check there is a clear preference for implicit waits, selected by 62.5% of participants. The second alternative is the use of explicit waits (12.5%). The third option is implementing manual wait strategies (e.g., `Thread.sleep()` in Java) with a quota of 11.11%. Then, we find no wait strategy (9.72%), and finally, fluent waits (4.17%).

The last question of this section is related to the assertion within the Selenium tests. In light of the results, the assertion strategy is quite heterogeneous. To get a clearer picture of the results, Figure 8 illustrates the reported assertions grouped by the different Selenium language bindings (Java, Python, JavaScript, C#, Ruby, Groovy, and PHP). In Java, the most relevant assertion technologies are directly related to the underlying unit testing framework: JUnit and TestNG assertions, with a relative percentage usage (respect to the total participants using Java as its preferred language) of 52.78% and 38.89% respectively. Concerning Python, the preferred assertion method is the native modules for assertions, with a quota of 76.92%. Similarly, we highlight the native assertions module in Node.js when using JavaScript as language binding, with a 40% quota. When coming to C# as language binding, we find NUnit assertions and Fluent Assertions, with 44.44% and 33.33%, respectively. In Ruby, there are different options in the two provided answers: RSpec::Expectations and Test::Unit Assertions. Finally, we find Hamcrest for Groovy and PHPUnit assertions for PHP.

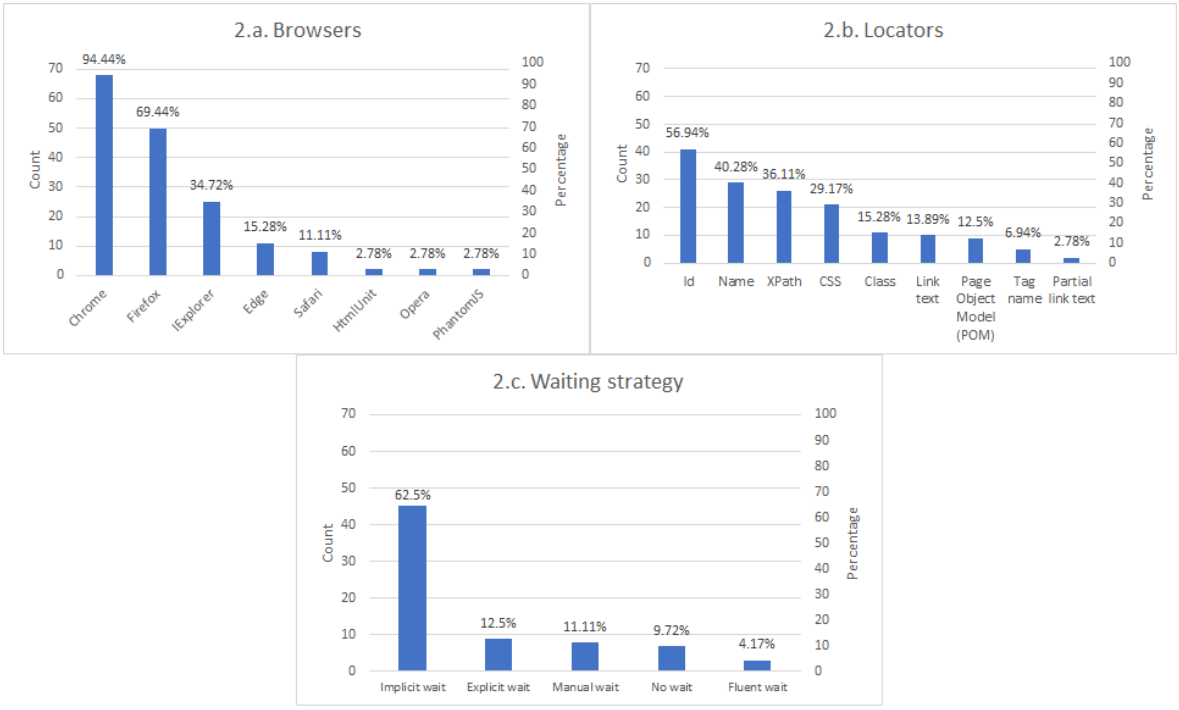

Figure 7. Web browser usage, locator, and waiting strategies.

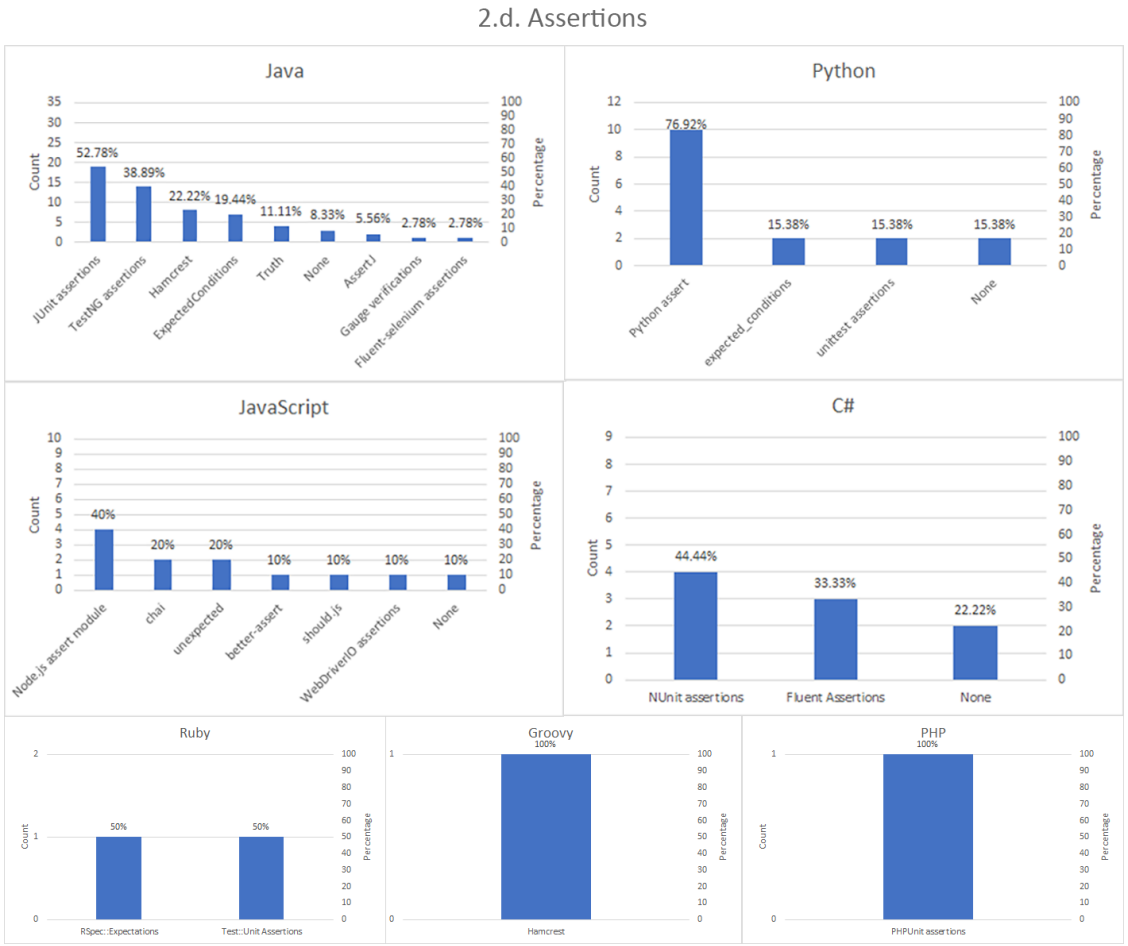

Figure 8. Assertion strategy.

4.3. System under Test

When coming to the third section of the survey about the SUT, we find the following. Regarding the application type, as shown the left chart of Figure 9, all the participants declared to test web applications (100%). Besides, 20.83% of the respondents reported testing also mobile applications (using Appium, as shown in Section 4.5). A small percentage of participants chose other options: browser extensions and wearables, both with 1.39%. Regarding the business sector, there is a clear preferred option, the enterprise applications, with a quota of 81.94%. The second and third options are e-commerce and real-time communications (WebRTC), with 18.06% and 5.56%, respectively. Comparatively, the other options have a minor impact: finance, games, maps, educational, graphics, software testing, Software as a Service (SaaS) applications, searching applications, social networking, and finally, sports. Regarding quality attributes, functional testing is the majority option, chosen by 98.61% of the respondents. Then we find performance (20.83%), usability (19.44%), compatibility (9.72%), accessibility (8.33%), and security (6.94%).

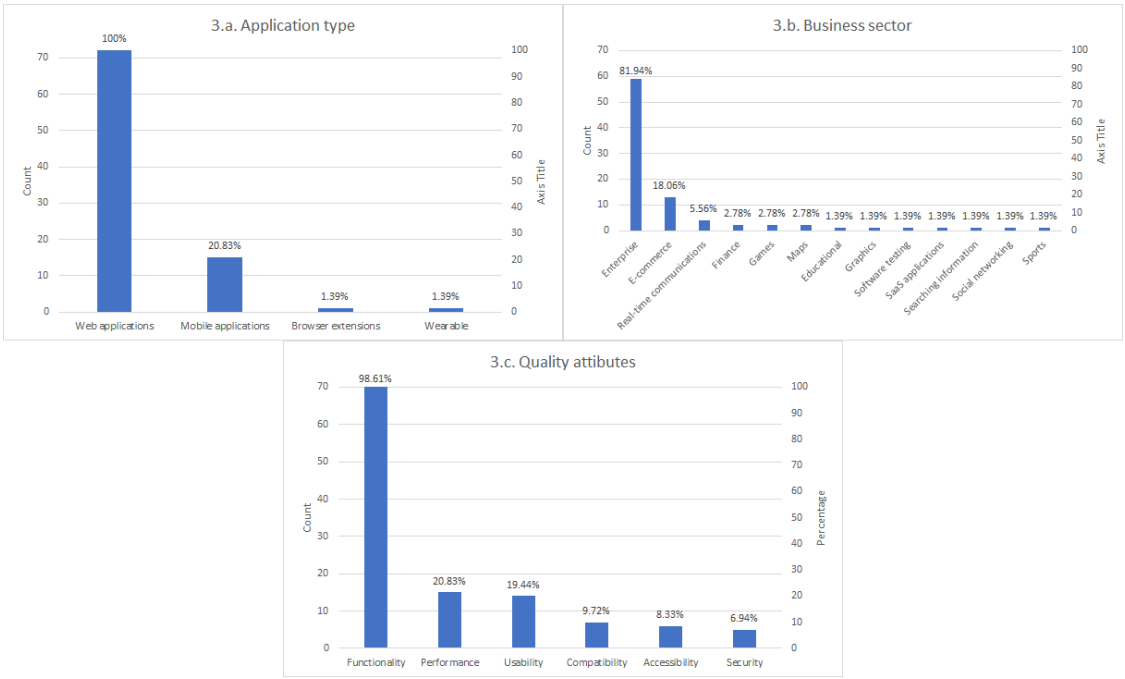

Figure 9. Application type, business sector, and quality attributes.

4.4. Test Infrastructure

The fourth section of the survey is about the infrastructure for Selenium tests. In this section, we asked a question about the CI server used in conjunction with Selenium. In this aspect, as shown in the left chart of Figure 10, Jenkins CI (https://www.jenkins.io/) is the preferred option, selected the 72.22% times by the respondents. Regarding the provider for the browser used by Selenium, we can categorize the answers in different groups:

- Local browsers, that is, installed in the machine running Selenium. This option is preferred by the respondents, with a total percentage of 59.72%.
- Browsers in Docker containers (selected by 38.89% of respondents). Within this option, 46.67% of respondents (relative to the total participants using Docker) use the official Docker images for browsers created by the Selenium team in the docker-selenium project (https://github.com/SeleniumHQ/docker-selenium). Zalenium (https://opensource.zalando.com/zalenium/) was chosen by 20% of participants using Docker. Zalenium is an extension to Selenium Grid based on Docker. The third Docker tool we find is Selenoid (https://aerokube.com/selenoid/), with a

relative quota of 16.67%. Selenoid is a high-performance implementation of Selenium Grid in Go language, which allows launching web browsers in Docker containers.

- Commercial on-demand browsers. In this option, we find SauceLabs (https://saucelabs.com/) with a quota of 18.06%, BrowserStack (https://www.browserstack.com/) with 13.89%, and CrossBrowserTesting (https://crossbrowsertesting.com/) with 1.39%.
- Remote browsers served by a custom Selenium Grid. A percentage of 18.06% of the respondents chose this option.

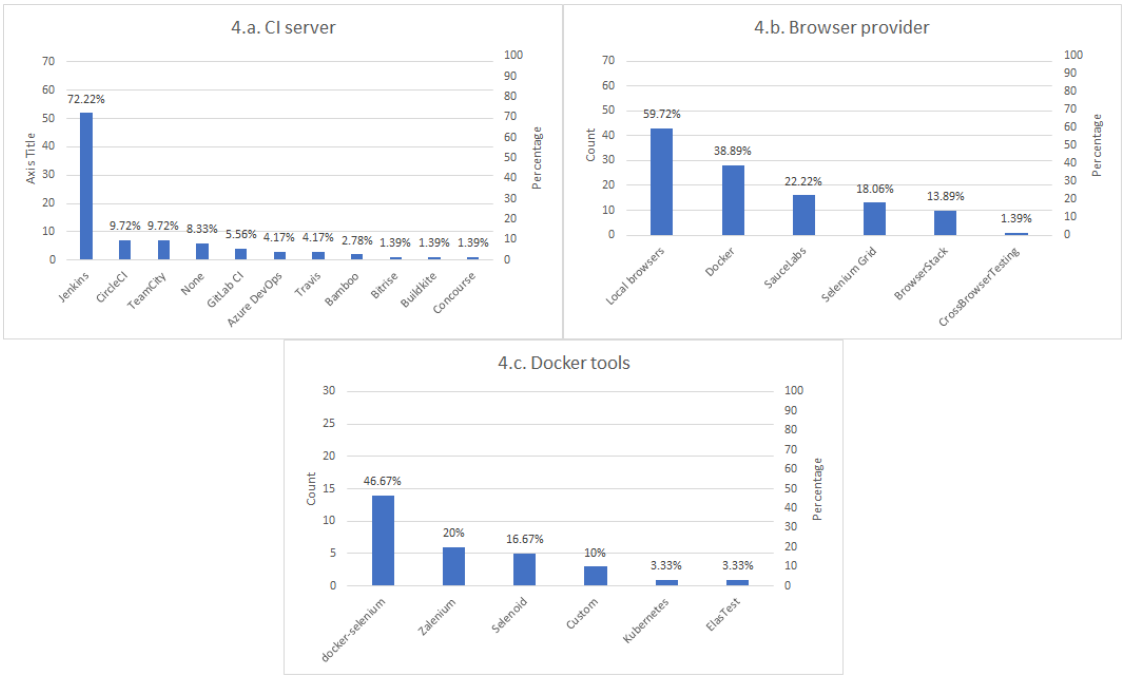

Figure 10. Infraestrucutre for Selenium tests.

4.5. Testing Frameworks

The fifth section of the survey is about other testing frameworks used together with Selenium. This first question of this category is about unit testing frameworks. We found a relevant number of different answers to this question in the survey. For the shake of clarity, the selected unit frameworks by respondents are grouped by the programming language in Figure 11. In Java, we check that there are two major unit frameworks: JUnit (75%) and TestNG (52.78%). In Python, the preferred option is PyTest (61.54%). Mocha was selected by half of the participants using JavaScript. NUnit is the preferred option by C# users (66.67%). The two respondents who declared to use Ruby used RSpec and Test::Unit as unit frameworks, respectively. In Groovy, the only respondent uses two different unit frameworks: JUnit and Spock. And finally, PHPUnit is used by the unique person using PHP in this survey.

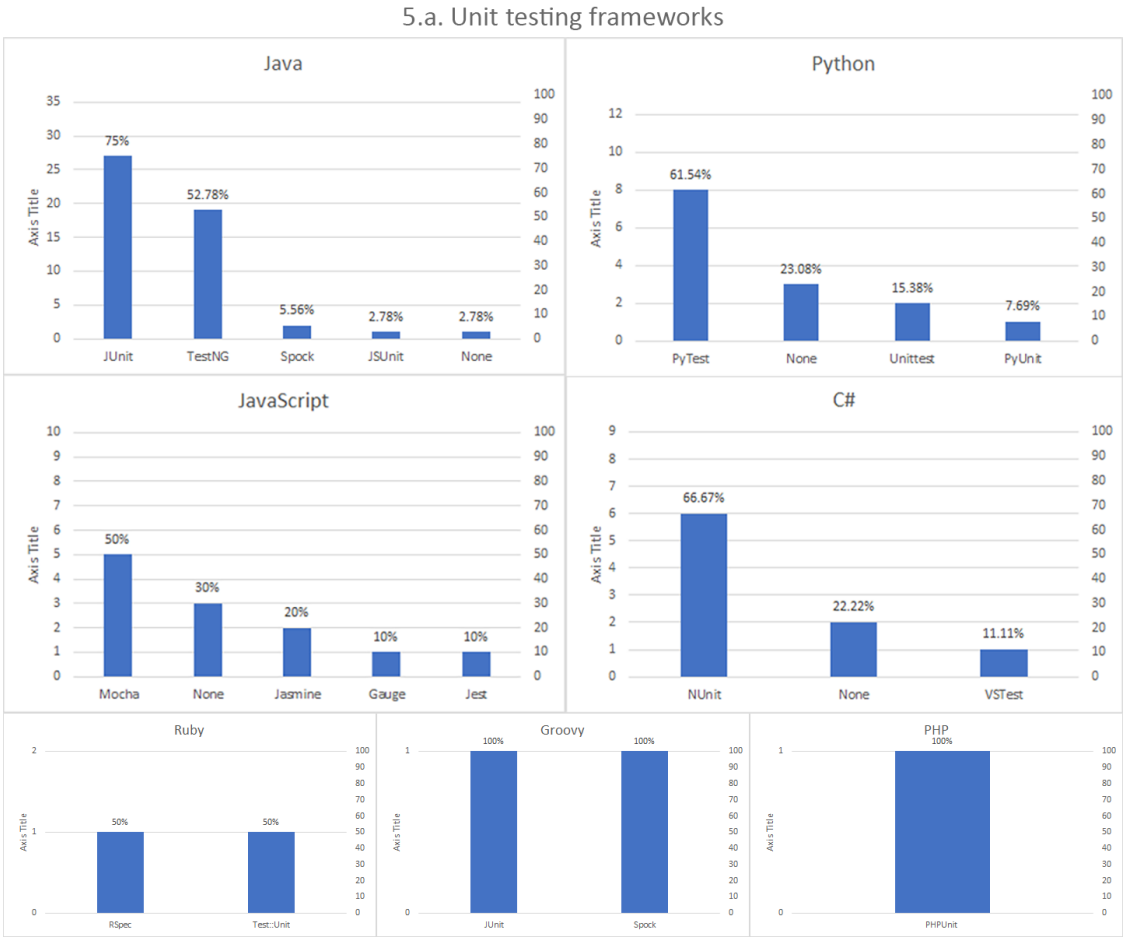

Figure 11. Unit testing frameworks used in conjunction with Selenium.

Regarding high-level frameworks based on or extending Selenium, the situation is also heterogeneous. As illustrated in Figure 12, we find several frameworks for different languages, namely:

- Cucumber (https://cucumber.io/) is a testing framework that allows writing acceptance tests following a Behavior-Driven Development (BDD) approach. Cucumber seems to be very relevant in the Selenium community since it was selected by the 36.11% of the respondents using Java as the primary language, 7.69% in Python, 40% in JavaScript, 22.25% in C#, and the 100% of Groovy and PHP users.
- Appium (http://appium.io/) is a testing framework that extends Selenium to carry out automated testing of mobile applications. Appium is also quite relevant in light of the survey results used by in Java (30.56%), Python (15.38%), JavaScript (10%), and PHP (100%) respondents.
- Zalenium. As introduced previously, Zalenium is a Selenium wrapper based on Docker. Zalenium was selected by respondents using Java (11.11%) and JavaScript (20%).

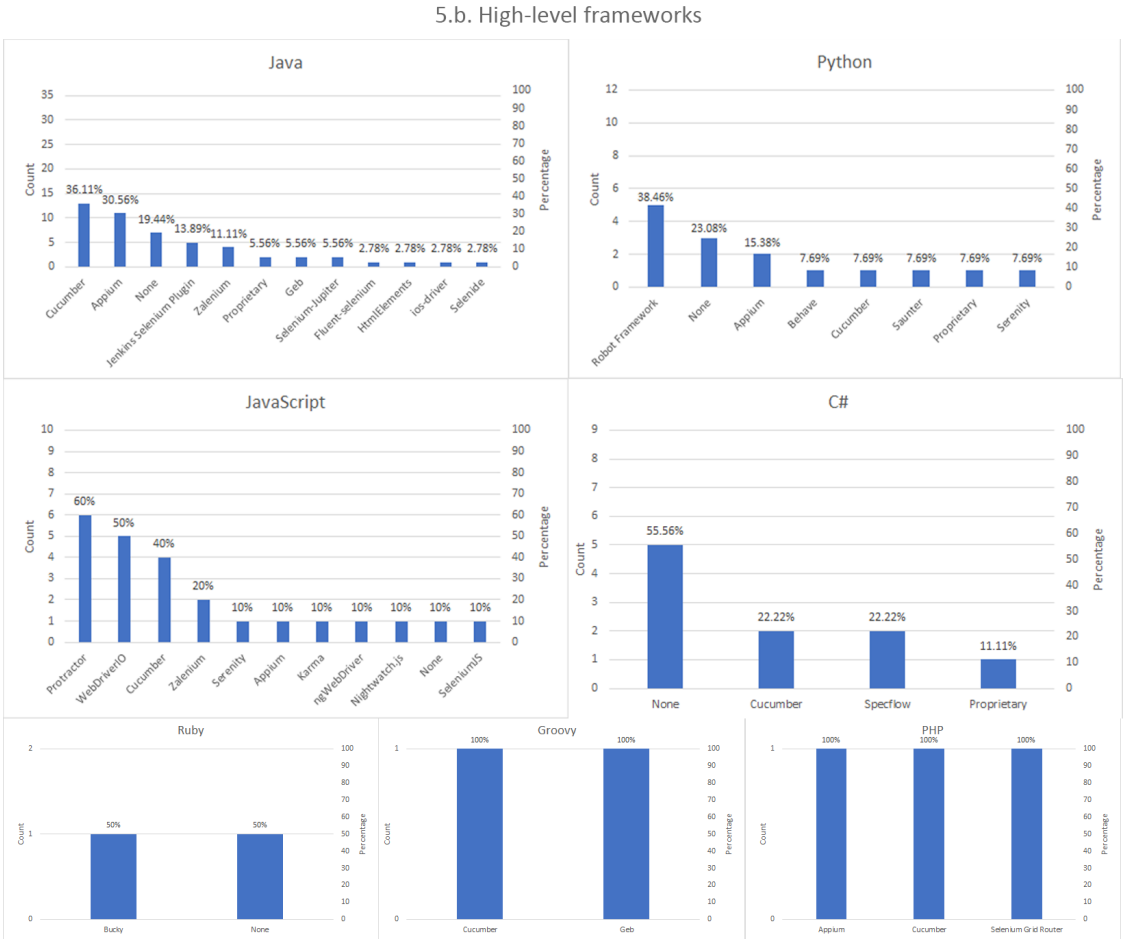

Figure 12. High-level testing frameworks based on Selenium.

In the third question of this section, we study the reporting tools used with Selenium. The top chart of Figure 13 shows these results. An option stands out respect to the others: none, with a quota of 62.5%. This option means that no report tools are used, and simply the test output is used instead. The second option is Allure (http://allure.qatools.ru/), with 11.11%. It is relevant to mention that this tool was selected even using different language bindings: Java, JavaScript, and Python.

The fourth question of this section is about the use of driver managers for Selenium WebDriver. As shown in the pie chart of Figure 13, 38.89% of the respondents declared to manage driver managers manually, while 34.72% of the respondents claimed to manage such drivers automatically. Besides, there is a significant percentage of users (26.39%) declaring not knowing how drivers are managed in their tests. The last chart in Figure 13 highlights the tools used for automated driver management. The first option is WebDriverManager (https://github.com/bonigarcia/webdrivermanager), with a usage quota of 22.22%. The second relevant option is the use of Docker containers for browsers, in which drivers are packaged inside the container.

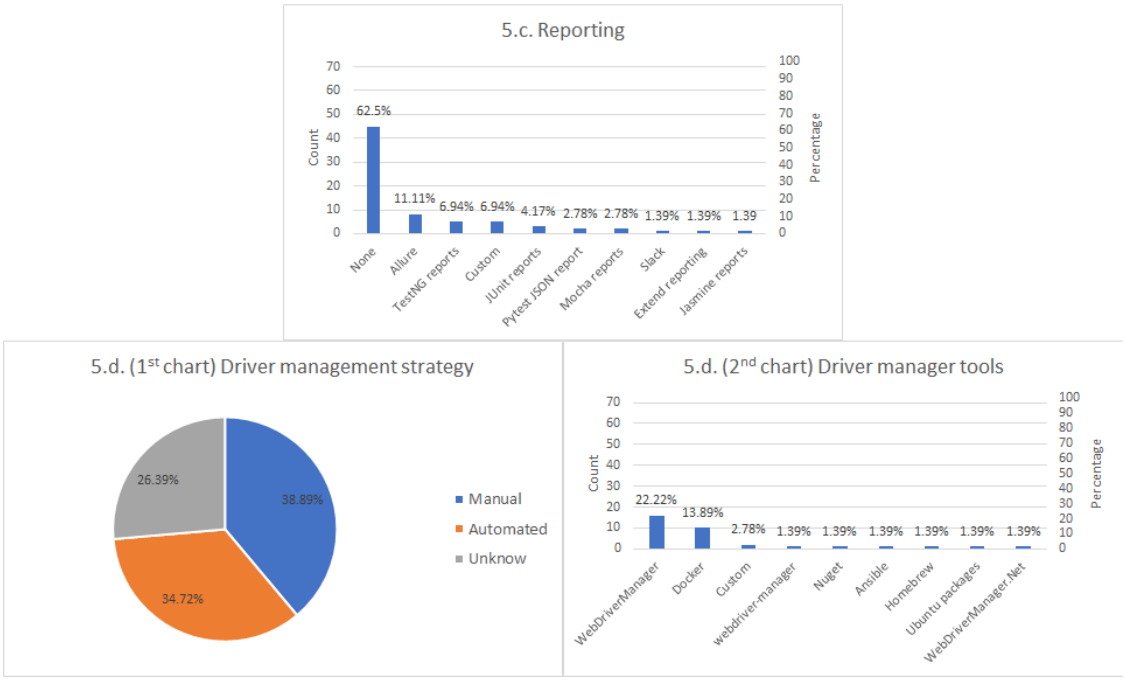

Figure 13. Reporting tools and driver managers for Selenium WebDriver.

The last question of this section focuses on AI/ML approaches. Figure 14 summarizes the results for this answer. As shown in the left chart of this picture, only 5.56% of the respondents declared to use AI/ML tools in conjunction with Selenium. These tools are Applitools (https://applitools.com/) (4.17%), a visual management AI-powered software for testing and monitoring, and test.ai (https://www.test.ai/) (1.39%), an AI-powered test platform for mobile testing.

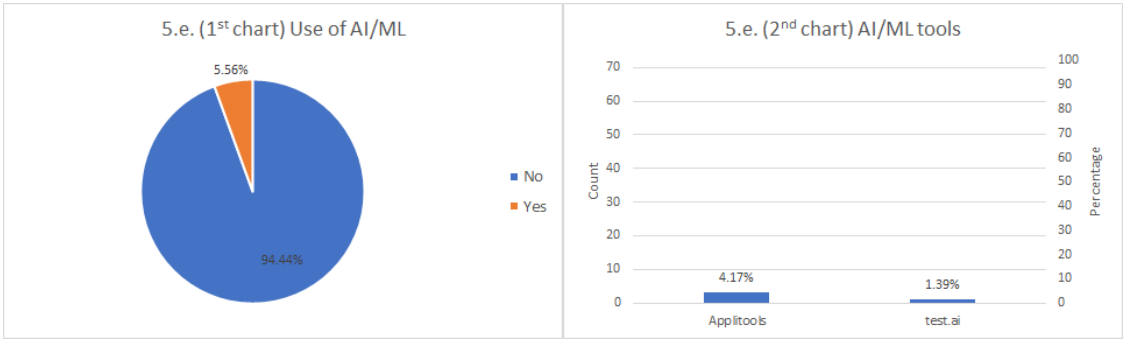

Figure 14. Artificial Intelligence (AI) and Machine Learning (ML) approaches used with Selenium.

4.6. Community

Figure 15 shows the results concerning the Selenium community. The left chart of this figure shows the respondents' preferred sources of information for Selenium documentation. The top information sources are online resources, namely online tutorials (43.06%), blogs (30.56%), and the official Selenium documentation (25%). Then we find books about Selenium (20.83%), followed by Massive Open Online Courses (MOOCs, 12.5%), Twitter (9.72%), webinars (5.56%), and finally podcasts (2.78%). Regarding Selenium support, most participants opted for StackOverflow (72.22%), followed by GitHub (30.56%). The rest of the selected options are the Selenium User Group (13.89%), Selenium's Slack channel (13.89%), and the Selenium Internet Relay Channel (IRC, 2.78%). The third question is about the events around Selenium, being SeleniumConf (https://www.seleniumconf.com/) and Selenium Meetups (https://www.meetup.com/topics/selenium/) the most relevant, selected 62.5% and 47.22% participants of this survey, respectively. Then, we find other events, namely TestBash (https://www.

ministryoftesting.com/testbash), SauceCon (https://saucecon.com/), Automation Guild Conference (https://guildconferences.com/), Heisenbug (https://heisenbug.ru/en/), and SeleniumCamp (https://seleniumcamp.com/).

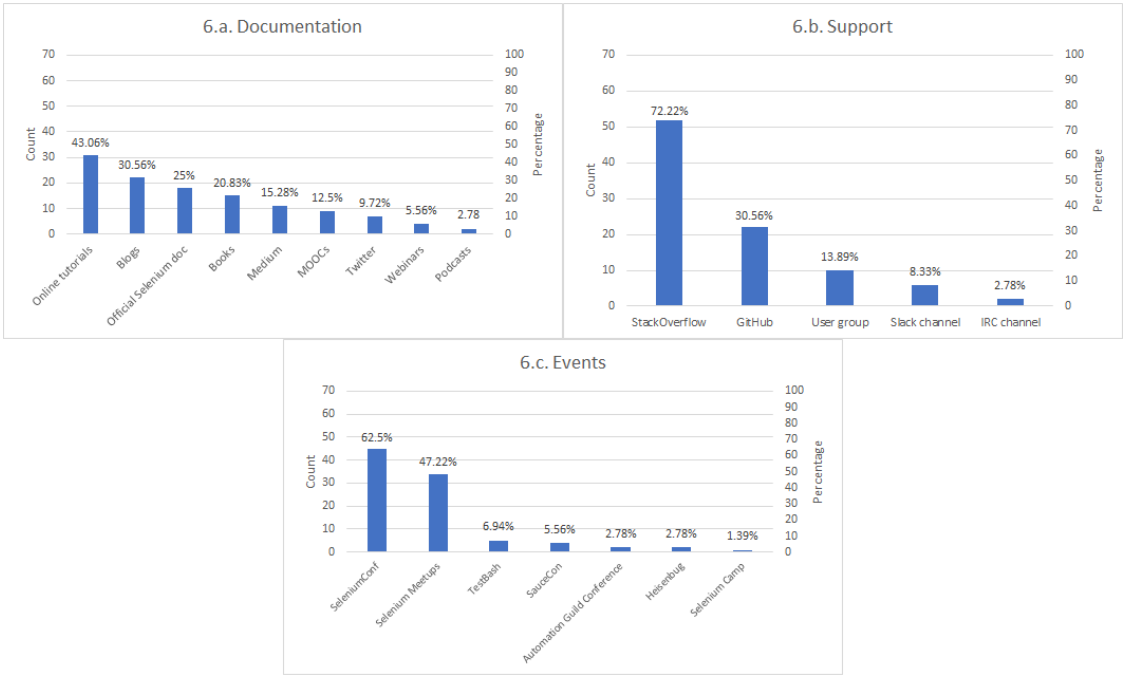

Figure 15. Sources of information for Selenium documentation, support, and events.

4.7. Personal Experience

This sixth and last part of the survey is devoted to open-ended answers based on the respondents' personal experience with Selenium. First, we ask to report the common problems found when using Selenium. As shown in Figure 16, we classify the answers to this question using the following categories:

- Maintainability (45.83%). The problem reported more times by respondents is related to the difficulties in the maintenance of Selenium tests. Respondents highlight the amount of time and effort (and, as a result, cost) as a critical aspect when maintaining a Selenium test suite.
- Flakiness (44.44%). A significant number of respondents declare than Selenium tests sometimes are flaky, that is, unreliable. In other words, the same Selenium test sometimes fails or passes for the same configuration [33].
- Test development (36.11%). In this category, we found different answers related to the creation of Selenium tests. Answers grouped in this category include answers with the sentences "difficult to create", "hard to write", "difficult to impersonate users", and "difficult to reuse". Selenium tests require an in-depth knowledge of how the different web pages under test are structured to use the proper locators and waits. This fact poses some difficulty to newcomers when trying to build selenium tests.
- Slowness (29.17%). Several participants declare the high execution time as a problem of Selenium tests.
- Troubleshooting (18.06%). When exercising the SUT through a browser, monitoring and log retrieval tools might be needed to understand what is happening in the SUT. Several respondents reported difficulties due to the lack of isolation of Selenium tests. In other words, when a Selenium test fails, it is not trivial to find the underlying error in the SUT.
- Non-functional assessment (15.28%). The infrastructure required to carry out load testing using Selenium (i.e., using a high number of web browser) were reported as challenging by some of the

participants. Usability and security were also mentioned by participants to be difficult to evaluate with Selenium.

- Integration (6.94%). Several participants declared integration issues related to Selenium. For some users, it would be desirable a more seamless integration with Appium or Jenkins.
- Contribution (1.39%). One of the respondents highlights difficulties contributing to the master branch of Selenium WebDriver due to the code complexity.

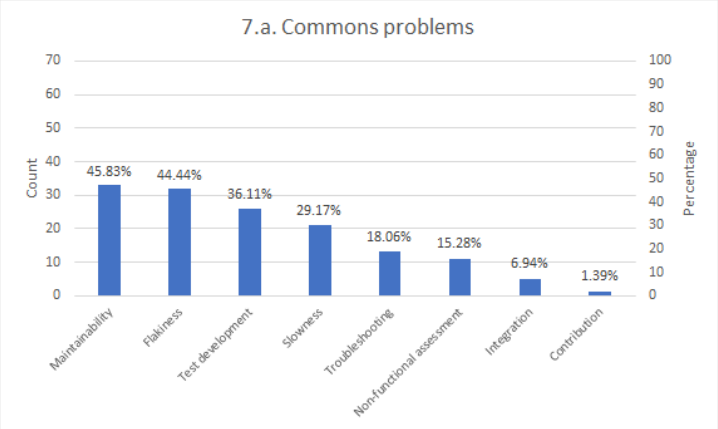

Figure 16. Categories for common problems related to Selenium.

The last question of this section is related to further comments about the Selenium ecosystem. In general, we check that the participants highly value Selenium. Besides, some of the respondents highlight the knowledge of Selenium as a differential factor in finding a job. The standardization of the W3C WebDriver is also mentioned as a relevant achievement.

5. Discussion

This section analyses the results of the survey. Regarding the Selenium projects, the first conclusion we can draw is that Selenium WebDriver is undeniably the heart of Selenium since almost all Selenium users utilize it. Regarding language binding, there is a preference for Java. In a second-tier, we find Python and JavaScript.

Regarding browsers, Chrome is the browser used for almost every Selenium tester, followed by Firefox. Surprisingly, the third browser in terms of Selenium automation is Internet Explorer, selected by 34.72% of the participants. Even though it is a deprecated browser nowadays, it seems it is still in use by the community. This fact can be related to the results of the SUT. There is a clear preference for testing the functional features of enterprise web applications. Internet Explorer was the dominant browser years ago, and it seems it is still in use in legacy Windows systems. All in all, practitioners need to ensure that Explorer still supports their applications, and Selenium is the selected tool to carry out this verification process.

Regarding infrastructure, there is a clear preference to use Jenkins as a CI server. Then, we find that the usage of local browsers is the preferred option. Nevertheless, we can see that Docker is also very relevant to Selenium. As explained in the next section, we can see another proof of the importance of Docker in Selenium in the features of the next version of Selenium Grid.

We interpret the diversity in the answers for several aspects studied by the survey as a health symptom of the Selenium ecosystem. First, the number of high-level frameworks based in Selenium is remarkable. This figure is a clear indicator of the success of Selenium as the germ to further technologies, such as Appium, Zalenium, or Robot Framework, to name a few. Moreover, we find other frameworks that are used in conjunction with Selenium to extend their original features. In this arena, Cucumber is a relevant actor, since the BDD approach used in Cucumber tests, can be used to implement end-to-end tests to assess web applications with Selenium.

Analyzing the results of the frameworks section, we can also identify potential weaknesses in the Selenium ecosystem, first, regarding reporting tools. In this domain, most of the respondents declared not using any reporting tool. A plausible explanation is that the plain output provided by the unit testing frameworks (i.e., red/green reports and error traces) is sufficient for most of the projects. Nevertheless, this fact might also be an indicator of a lack of comprehensive specific reporting tools for Selenium. This view is reinforced by the fact that some respondents reported that it is a challenge to troubleshoot problems when tests fail. Proper reporting tools could bring visibility over the whole test infrastructure. In this arena, Allure (the second choice in light of results) might be a candidate to fulfill this need, since it can be used from different binding languages in Selenium. Another improvement point is the use of driver managers. A relevant percentage of respondents declared to manage driver manually, which in the long run, might cause maintainability and test development problems. Tools like WebDriverManager and other managers promise to solve this problem, but it seems the usage of this kind of helper utility is not wide-spread yet. Finally, although the use of AI/ML techniques promises to revolutionize the testing arena, at the time of this writing, there is still limited adoption of such methods (only Applitools and test.ai were reported in the survey).

We can get some conclusions from the numbers about the community. Regarding the documentation, it is remarkable that the two preferred options are online tutorials and blogs. These options are, in general, maintained by the community. Again, this is an indicator or the interest in Selenium. This fact is endorsed by support results, in which another community-based solution, StackOverflow, is the preferred option to get support about Selenium. Regarding the events, the most significant events are the official conference (SeleniumConf) and Meetup. Nevertheless, it is interesting to discover that other events, not oriented explicitly to Selenium but for Quality Assurance (QA) and testing (such as TestBash or SauceCon), are also identified as related to Selenium.

Concerning the common problems of Selenium, the first group of difficulties is about the maintainability of Selenium tests. As introduced in Section 2.1, maintainability is a well-known issue in test automation in general and Selenium in particular. Nevertheless, this problem should not be a stopper in the adoption of test automation since the maintenance cost of automated tests is proved to provide a positive return on investment compared to manual testing [34]. When coming to Selenium, the maintenance and evolution of end-to-end tests (especially large-scale suites), to be efficient, requires the adoption of appropriate techniques and best practices [35]. To this aim, the adoption of design patterns in the test codebase is recommended. The test patterns proposed by Meszaros (such as the delegated test method setup or customs assertions, to name a few) can enhance the quality of the xUnit tests that embed the Selenium logic [36]. Then, and specific to Selenium, the usage of page objects is a well-known mechanism to improve the reusability, maintainability, and readability of Selenium tests. According to Leotta et al., the adoption of POM in Selenium tests contributes to significant effort reduction: maintenance time is reduced in a factor of 3, and lines of code (LOCs) to repair tests is reduced in a factor of 8 [26]. The fact that maintainability is the first category of problems pointed out by almost half of the surveyed people is in line with our finding that few respondents use page objects to model their web page interactions. In light of the evidence reported in previous literature, we think that the adoption of POM could drastically increase the reliability of Selenium tests. To ease the adoption of POM, Selenium WebDriver provides a page factory utility set [37]. Within the scope of POM, we find another pattern called Screenplay, which is the application of the SOLID design principles to page objects. The Screenplay pattern uses actors, tasks, and goals to define tests in business terms rather than interactions with the SUT [38].

The locator strategy in Selenium tests is also reported as a possible cause of maintainability issues. We consider that practitioners need to be aware of this potential problem to prevent it, designing a coordinated plan between the front-end and testing teams aimed to ease and maintain the testability during the project lifecycle. In this regard, different strategies can be adopted. For instance, Leotta et al. investigate the LOC modified and the time required to evolve Selenium WebDriver

tests when using different locator strategies. Although not conclusive, this work reported fewer maintenance efforts when using id locators in conjunction with link text compared to XPath [32]. The automatic generation of locator strategies has also been reported in the literature to avoid fragility in Selenium tests. For example, Reference [39] proposes an algorithm called ROBULA (ROBUst Locator Algorithm) to automatically generated locators based on XPath. This algorithm iteratively searches in the DOM nodes until the element of interest is found. To improve the reliability of this approach, the authors of this work propose a multi-locator strategy that selects the best locator among different candidates produced by different algorithms [40].

Another very relevant common problem detected is the unreliability of some tests, often known as flaky. Generally speaking, the first step to fix a flaky test is to discover the cause. There are different causes of non-deterministic tests in Selenium, including fragile location strategies, inconsistent assertions, or incorrect wait strategies. Again, we think that using the POM model together with robust waiting and location strategies could increase the reliability of flaky tests. Regarding waiting strategies, as we saw, that there is a preference for implicit waits. Nevertheless, several users declared no using wait strategies or manual waits. Depending on the hardware and the SUT, this fact might lead to flaky tests [41]. Regarding troubleshooting, observability tools like the ELK (ElasticSearch, Logstash, and Kibana) stack [42] or ElasTest (https://elastest.io/) [43] could help alleviate this problem.

5.1. Evolution of Selenium and Alternatives

The immediate future of Selenium is version 4, in alpha at the time of this writing. Selenium 4 brings new features for all the core project of Selenium. A key point in Selenium WebDriver 4 is the full adoption of the standard W3C WebDriver (https://www.w3.org/TR/webdriver/) specification as the only way to automate browsers. In other words, the communication between Selenium and a browser driver is entirely standard as of Selenium WebDriver 4. This adoption is expected to bring stability to the browser automation through Selenium since a standards committee maintains the automation aspects which browser vendors should adopt. Another new feature in Selenium WebDriver 4 is the introduction of relative locators, that is, the capability to find nearby other elements or the exposure of the DevTools API for Chromium-based drivers (continue reading this section for further details about DevTools). Version 4 also brings novelties to the rest of the official Selenium family. When coming to the Selenium Grid project, the Selenium server (hub) performance and its user interface are highly improved in the new version. Besides, Selenium Grid incorporates native Docker support for nodes and supports GraphQL for making queries. Regarding Selenium IDE, the next version includes a Comand-Line Interface (CLI) runner, and the ability to specify different locator strategies when exporting test cases. Last but not least, the official Selenium documentation has been renewed.

To conclude, we consider it is interesting to review alternatives to Selenium in the browser automation arena. A relevant project to be considered is Puppeteer (https://pptr.dev/), a Node.js library created by Google that allows controlling browsers based on the Blink rendering engine (i.e., Chrome, Chromium, Edge, or Opera) over the DevTools Protocol. DevTools is a set of developer tools built directly into Blink-based browsers. DevTools allows editing web pages on-the-fly, debugging CSS and JavaScript, or analyzing load performance, among other features. The DevTools Protocol (https://chromedevtools.github.io/devtools-protocol/) is a protocol based on JSON-RPC messages, which allows instrumenting, inspecting, debugging, and profiling Blink-based browsers.

Playwright (https://playwright.dev/) is also a Node.js library for browser automation created by Microsoft. The Playwright team is made up of ex-Googlers who previously worked developing Puppeteer. We can say that Playwright is a newcomer in the browser automation space since the first public release of Playwright was in January 2020. Although the API of Playwright and Puppeteer are quite similar, internally, both tools are quite different. One of the most significant differences between Playwright and Puppeteer is cross-browser support. In addition to Chromium-based browsers, Playwright can drive WebKit (the browser engine for Safari) and Firefox. To implement this

cross-browser support, Playwright is shipped with patched versions of WebKit and Firefox in which their debugging protocols are extended to support remote capabilities through the Playwright API.

Cypress (https://www.cypress.io/) is another Node.js framework that allows implementing end-to-end tests using a sandbox environment based on JavaScript following a client-server architecture. Tests on Cypress runs inside the browser, and therefore the waits for web elements are automatically managed by Cypress. The limitations of Cypress are the lack of cross browsing (for example, Safari is not supported) and same-origin constraints (i.e., to visit URLs with different protocol and host in the same test).

Sikuli (http://www.sikulix.com/) is a tool to automate the interaction within a desktop environment from Windows, Mac, or Linux/Unix. It uses image recognization based on OpenCV to identify GUI components [44]. It supports several scripting languages, including Python, Ruby, JavaScript, Robot Framework text-scripts, and Java-aware scripting (e.g., Jython, JRuby, Scala, or Clojure).

5.2. Threats to Validity

We discuss the main threats to the validity of our research following commonly accepted techniques described in Reference [45]. This way, we analyze the construct, internal, and external validity.

We carried out a careful design of our questionnaire using high-level dimensions of the Selenium ecosystem to minimize threats in the construct. For the shake of completeness, we include the "other" field to allow custom answers in many of the questions. Finally, and before launching the survey, we did a thorough review of the questionnaire with the help of several experts in testing with Selenium.

Internal validity is the extend in which the design of the experiment avoids introducing bias into the measurement. To enhance our internal validity, we tried to avoid any selection bias by enabling the Selenium community to answer the questionnaire freely. This strategy was successful, given the number and spectrum of participants in terms of age and nationality. The risk of statistical effects in the outcome (for non-language-specific questions) is low since 72 participants completed the questionnaire, which is a significant population for this kind of survey.

External validity refers to the extent to which the results can be generalized. In this regard, the main threat comes from a lack of statistical significance. We designed the survey to get a snapshot of the Selenium ecosystem, but no hypotheses were defined at the beginning of the process. This fact, together with the low rate of some language bindings (e.g., C#, Ruby, Groovy, and PHP), poses a generalizability problem.

6. Conclusions

The Selenium framework (made up of the projects WebDriver, Grid, and IDE) is considered by many as the de facto standard for end-to-end web testing. It has come a long way since its inception in 2004. Nowadays, the Selenium ecosystem comprises a wide range of variety of other frameworks, tools, and approaches available around the root Selenium projects.

This paper presents an effort to investigate the Selenium ecosystem. To this aim, we launched to the Selenium community a descriptive survey in 2019. In the light of results, we can see that the average Selenium tester uses Selenium WebDriver and Jenkins to drive Chrome automatically and assess the functional features of enterprise web applications. Nevertheless, in addition to this common usage, we check there is a wide variety of other elements in the Selenium ecosystem, including assertion libraries or high-level frameworks (for example, Appium or Cucumber). Besides, we discovered that Docker is already a relevant actor in Selenium. However, there is still room to generalize the use of additional parts of the ecosystem, such as driver managers (e.g., WebDriverManager) or reporting tools (e.g., Allure). The adoption of AI/ML approaches (e.g., Applitools) is still at an early stage.

We believe the results presented in this paper help to provide a deeper understanding of the development of end-to-end tests with Selenium and the relationship with its ecosystem.

Nevertheless, there are still open challenges that deserve further attention. In particular, we think that the reported problems related to Selenium might outline future research directions in the browser automation space. To this aim, additional effort should be made to identify the root causes of the commons problems related to Selenium tests. In this arena, future research might help to determine the relationship of the identified problems (such as maintainability, flakiness, or troubleshooting) with specific aspects, such as the use of design patterns (e.g., POM), waiting and location strategies, or the adoption of observability mechanisms.

Finally, another aspect that might drive further research is the assessment of non-functional attributes with Selenium. Although Selenium is a framework mostly used for functional testing, our survey reveals that some practitioners also carry out non-functional testing with it. We think that future investigations about how to perform, for example, security, usability, or accessibility assessment with Selenium, might reveal novel approaches in the end-to-end testing domain.

Author Contributions: Conceptualization, B.G.; Data curation, B.G.; Funding acquisition, M.G., F.G. and M.M.-O.; Investigation, B.G.; Methodology, B.G.; Project administration, M.G., F.G. and M.M.-O.; Validation, B.G.; Visualization, B.G.; Writing—original draft, B.G.; Writing—review & editing, B.G., M.G., F.G. and M.M.-O. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the European Commission under the H2020 project "MICADO" (GA-822717), by the Government of Spain through the project "BugBirth" (RTI2018-101963-B-100), by the Regional Government of Madrid (CM) through the project "EDGEDATA-CM" (P2018/TCS-4499) cofunded by FSE & FEDER, and by the project "Analytics using sensor data for FlatCity" (MINECO/ERDF, EU) funded in part by the Spanish Agencia Estatal de Investigación (AEI) under Grant TIN2016-77158-C4-1-R and in part by the European Regional Development Fund (ERDF).

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

Appendix A. Survey Raw Results

Appendix A.1. Foundations

Table A1. 1.a. Which Selenium project do you use?

Selenium Projects	Count
WebDriver	71
Grid	27
IDE	11

Table A2. 1.b. Which primary binding language do you use with Selenium?

Language Bindings	Count
Java	36
Python	13
JavaScript	10
C#	9
Ruby	2
Groovy	1
PHP	1

Appendix A.2. Test

Table A3. 2.a. Which browser do you automate with Selenium?

Browsers	Count
Chrome	68
Firefox	50
IExplorer	25
Edge	11
Safari	8
HtmlUnit	2
Opera	2
PhantomJS	2

Table A4. 2.b. What locator strategy do you implement in your Selenium tests?

Locators	Count
Id	41
Name	29
XPath	26
CSS	21
Class	11
Link text	10
POM	9
Tag name	5
Partial link text	2

Table A5. 2.c. What wait strategy do you implement in your Selenium tests?

Waiting	Count
Implicit wait	45
Explicit wait	9
Manual wait	8
No wait	7
Fluent wait	3

Table A6. 2.d. What kind of assertions do you implement in your Selenium tests?

Language	Assertions	Count
Java	JUnit assertions	19
	TestNG assertions	14
	Hamcrest	8
	ExpectedConditions	7
	Truth	4
	None	3
	AssertJ	2
	Gauge verifications	1
	Fluent-selenium assertions	1
Python	Python assert	10
	expected_conditions	2
	unittest assertions	2
	None	2
JavaScript	Node.js assert module	4
	chai	2
	unexpected	2
	better-assert	1
	should.js	1
	WebDriverIO assertions	1
	None	1
C#	NUnit assertions	4
	Fluent Assertions	3
	None	2
Ruby	RSpec::Expectations	1
	Test::Unit Assertions	1
Groovy	Hamcrest	1
PHP	PHPUnit assertions	1

Appendix A.3. SUT

Table A7. 3.a. What is the type of your system under test?

Application Type	Count
Web applications	72
Mobile applications	15
Browser extensions	1
Wearable	1

Table A8. 3.b. In which sector do you classify your system under test?

Business Sector	Count
Enterprise	59
E-commerce	13
Real-time communications	4
Finance	2
Games	2
Maps	2
Educational	1
Graphics	1
Software testing	1
SaaS applications	1
Searching information	1
Social networking	1
Sports	1

Table A9. 3.c. Which type of test(s) do you implement?

Quality Attributes	Count
Functionality	71
Performance	15
Usability	14
Compatibility	7
Accessibility	6
Security	5

Appendix A.4. Infrastructure

Table A10. 4.a. Which CI server do you use?

CI	Count
Jenkins	52
CircleCI	7
TeamCity	7
None	6
GitLab CI	4
Azure DevOps	3
Travis	3
Bamboo	2
Bitrise	1
Buildkite	1
Concourse	1

Table A11. 4.b. Which browser provider do you use for your Selenium tests?

Browser Provider	Count
Local browsers	43
Docker	28
SauceLabs	16
Selenium Grid	13
BrowserStack	10
CrossBrowserTesting	1

Table A12. 4.c. If you use Docker with Selenium, which tool do you use to manage the containers?

Docker	Count
docker-selenium	14
Zalenium	6
Selenoid	5
Custom	3
Kubernetes	1
ElasTest	1

Appendix A.5. Frameworks

Table A13. 5.a. Which unit testing framework do you use in conjunction with Selenium?

Language	Unit Testing	Count
Java	JUnit	27
	TestNG	19
	Spock	2
	JSUnit	1
	None	1
Python	PyTest	8
	None	3
	Unittest	2
	PyUnit	1
JavaScript	Mocha	5
	None	3
	Jasmine	2
	Gauge	1
	Jest	1
C#	NUnit	6
	None	2
	VSTest	1
Ruby	RSpec	1
	Test::Unit	1
Groovy	JUnit	1
	Spock	1
PHP	PHPUnit	1

Table A14. 5.e. Do you use some Artificial Intelligence (AI) and/or Machine Learning (ML) approach in your Selenium tests?

AI/ML	Count
Applitools	3
test.ai	1

Table A15. 5.b. Which high-level testing framework (which extends or are based on Selenium) do you use?

Language	High-Level	Count
Java	Cucumber	13
	Appium	11
	None	7
	Jenkins Selenium Plugin	5
	Zalenium	4
	Proprietary	2
	Geb	2
	Selenium-Jupiter	2
	Fluent-selenium	1
	HtmlElements	1
	ios-driver	1
	Selenide	1
Python	Robot Framework	5
	None	3
	Appium	2
	Behave	1
	Cucumber	1
	Saunter	1
	Proprietary	1
	Serenity	1
JavaScript	Protractor	6
	WebDriverIO	5
	Cucumber	4
	Zalenium	2
	Serenity	1
	Appium	1
	Karma	1
	ngWebDriver	1
	Nightwatch.js	1
	None	1
	SeleniumJS	1
C#	None	5
	Cucumber	2
	Specflow	2
	Proprietary	1
Ruby	Bucky	1
	None	1
Groovy	Cucumber	1
	Geb	1
PHP	Appium	1
	Cucumber	1
	Selenium Grid Router	1

Table A16. 5.c. What kind of reporting do you get from your Selenium tests?

Reporting	Count
None	45
Allure	8
TestNG reports	5
Custom	5
JUnit reports	3
Pytest JSON report	2
Mocha reports	2
Slack	1
Extend reporting	1
Jasmine reports	1

Table A17. 5.d. How do you manage the driver binaries (e.g., *chromedriver*, *geckodriver*) required by Selenium WebDriver?

Driver Managers	Count
WebDriverManager	16
Docker	10
Custom	2
webdriver-manager	1
Nuget	1
Ansible	1
Homebrew	1
Ubuntu packages	1
WebDriverManager.Net	1

Appendix A.6. Community

Table A18. 6.a. What are the sources of information to learn and keep updated about Selenium?

Documentation	Count
Online tutorials	31
Blogs	22
Official Selenium doc	18
Books	15
Medium	11
MOOCs	9
Twitter	7
Webinars	4
Podcasts	2

Table A19. 6.b. What is, in your experience, the most convenient ways to obtain support about Selenium from the community?

Support	Count
StackOverflow	52
GitHub	22
User group	10
Slack channel	6
IRC channel	2

Table A20. 6.c. What, in your opinion, are the most relevant events related to Selenium?

Events	Count
SeleniumConf	45
Selenium Meetups	34
TestBash	5
SauceCon	4
Automation Guild Conf	2
Heisenbug	2
Selenium Camp	1

Appendix A.7. Personal Experience

Table A21. 7.a. What are the common problems you face when testing with Selenium?

Common Problems	Count
Maintainability	33
Flakiness	32
Test development	26
Slowness	21
Troubleshooting	13
Non-functional assessment	11
Integration	5
Contribution	1

References

1. Niranjanamurthy, M.; Navale, S.; Jagannatha, S.; Chakraborty, S. Functional Software Testing for Web Applications in the Context of Industry. *J. Comput. Theor. Nanosci.* **2018**, *15*, 3398–3404. [CrossRef]
2. Cerioli, M.; Leotta, M.; Ricca, F. What 5 million job advertisements tell us about testing: A preliminary empirical investigation. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 1586–1594.
3. Jansen, S.; Cusumano, M.A.; Brinkkemper, S. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*; Edward Elgar Publishing: Cheltenham, UK, 2013.
4. Vila, E.; Novakova, G.; Todorova, D. Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats. In Proceedings of the International Conference on Advances in Image Processing, Bangkok, Thailand, 25–27 August 2017; pp. 144–150.
5. Stewart, S.; Burns, D. WebDriver. W3C Working Draft. 2020. Available online: https://www.w3.org/TR/webdriver/ (accessed on 29 June 2020).
6. Pietraszak, M. Browser Extensions. W3C Community Group Draft Report. 2020. Available online: https://browserext.github.io/browserext/ (accessed on 29 June 2020).
7. Kredpattanakul, K.; Limpiyakorn, Y. Transforming JavaScript-Based Web Application to Cross-Platform Desktop with Electron. In *International Conference on Information Science and Applications*; Springer: Berlin, Germany, 2018; pp. 571–579.
8. Rafi, D.M.; Moses, K.R.K.; Petersen, K.; Mäntylä, M.V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In Proceedings of the 2012 7th International Workshop on Automation of Software Test (AST), Zurich, Switzerland, 2–3 June 2012; pp. 36–42.
9. Kasurinen, J.; Taipale, O.; Smolander, K. Software test automation in practice: Empirical observations. *Adv. Softw. Eng.* **2010**, *2010*, 620836. [CrossRef]
10. Bures, M. Automated testing in the Czech Republic: The current situation and issues. In Proceedings of the 15th International Conference on Computer Systems and Technologies, Ruse, Bulgaria, 27–28 June 2014; pp. 294–301.
11. Bures, M.; Filipsky, M. SmartDriver: Extension of selenium WebDriver to create more efficient automated tests. In Proceedings of the 2016 6th International Conference on IT Convergence and Security, (ICITCS), Prague, Czech Republic, 26 September 2016; pp. 1–4.

12. Leotta, M.; Stocco, A.; Ricca, F.; Tonella, P. Pesto: Automated migration of DOM-based Web tests towards the visual approach. *Softw. Test. Verif. Reliab.* **2018**, *28*, e1665. [CrossRef]
13. Alegroth, E.; Nass, M.; Olsson, H.H. JAutomate: A tool for system-and acceptance-test automation. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–22 March 2013; pp. 439–446.
14. García, B.; López-Fernández, L.; Gortázar, F.; Gallego, M. Practical Evaluation of VMAF Perceptual Video Quality for WebRTC Applications. *Electronics* **2019**, *8*, 854. [CrossRef]
15. García, B.; Gortázar, F.; Gallego, M.; Hines, A. Assessment of QoE for Video and Audio in WebRTC Applications Using Full-Reference Models. *Electronics* **2020**, *9*, 462. [CrossRef]
16. García, B.; Gallego, M.; Gortázar, F.; Bertolino, A. Understanding and estimating quality of experience in WebRTC applications. *Computing* **2019**, *101*, 1585–1607. [CrossRef]
17. Christophe, L.; Stevens, R.; De Roover, C.; De Meuter, W. Prevalence and maintenance of automated functional tests for web applications. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 141–150.
18. Bures, M.; Filipsky, M.; Jelinek, I. Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts. *Int. J. Softw. Eng. Knowl. Eng.* **2018**, *28*, 3–36. [CrossRef]
19. Stocco, A.; Leotta, M.; Ricca, F.; Tonella, P. APOGEN: Automatic page object generator for web testing. *Softw. Qual. J.* **2017**, *25*, 1007–1039. [CrossRef]
20. Kuutila, M. Benchmarking Configurations for Web-Testing-Selenium Versus Watir. Master's Thesis, Faculty of Information Technology and Electrical Engineering, University of Oulu, Oulu, Finland, 2016.
21. Kaur, H.; Gupta, G. Comparative study of automated testing tools: Selenium, quick test professional and testcomplete. *Int. J. Eng. Res. Appl.* **2013**, *3*, 1739–1743.
22. Glass, R.L.; Ramesh, V.; Vessey, I. An analysis of research in computing disciplines. *Commun. ACM* **2004**, *47*, 89–94. [CrossRef]
23. Glass, R.L.; Vessey, I.; Ramesh, V. Research in software engineering: An analysis of the literature. *Inform. Softw. Technol.* **2002**, *44*, 491–506. [CrossRef]
24. Reja, U.; Manfreda, K.L.; Hlebec, V.; Vehovar, V. Open-ended vs. close-ended questions in web questionnaires. *Devel. Appl. Stat.* **2003**, *19*, 159–177.
25. Clark, J.; DeRose, S. XML Path Language (XPath) Version 1.0. W3C Recommendation. 1999. Available online: https://www.w3.org/TR/1999/REC-xpath-19991116/ (accessed on 29 June 2020).
26. Leotta, M.; Clerissi, D.; Ricca, F.; Spadaro, C. Improving test suites maintainability with the page object pattern: An industrial case study. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 18–22 March 2013; pp. 108–113.
27. Avasarala, S. *Selenium WebDriver Practical Guide*; Packt Publishing Ltd.: Birmingham, UK, 2014.
28. Barr, E.T.; Harman, M.; McMinn, P.; Shahbaz, M.; Yoo, S. The oracle problem in software testing: A survey. *IEEE Trans. Softw. Eng.* **2014**, *41*, 507–525. [CrossRef]
29. Orso, A.; Rothermel, G. Software testing: A research travelogue (2000–2014). In Proceedings of the on Future of Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 117–132.
30. Diouf, R.; Sarr, E.N.; Sall, O.; Birregah, B.; Bousso, M.; Mbaye, S.N. Web Scraping: State-of-the-Art and Areas of Application. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 6040–6042.
31. Khomh, F.; Adams, B.; Cheng, J.; Fokaefs, M.; Antoniol, G. Software engineering for machine-learning applications: The road ahead. *IEEE Softw.* **2018**, *35*, 81–84. [CrossRef]
32. Leotta, M.; Clerissi, D.; Ricca, F.; Spadaro, C. Comparing the maintainability of Selenium WebDriver test suites employing different locators: A case study. In Proceedings of the 2013 International Workshop on Joining Academia and Industry Contributions to Testing Automation, Lugano, Switzerland, 15–20 July 2013; pp. 53–58.
33. Luo, Q.; Hariri, F.; Eloussi, L.; Marinov, D. An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 643–653.
34. Alégroth, E.; Feldt, R.; Kolström, P. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Inform. Softw. Technol.* **2016**, *73*, 66–80. [CrossRef]

35. Garousi, V.; Felderer, M. Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Softw.* **2016**, *33*, 68–75. [CrossRef]
36. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*; Pearson Education: London, UK, 2007.
37. Kovalenko, D. *Selenium Design Patterns and Best Practices*; Packt Publishing Ltd.: Birmingham, UK, 2014.
38. Agarwal, A.; Gupta, S.; Choudhury, T. Continuous and Integrated Software Development using DevOps. In Proceedings of the 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), Paris, France, 22–23 June 2018; pp. 290–293.
39. Leotta, M.; Stocco, A.; Ricca, F.; Tonella, P. Reducing web test cases aging by means of robust XPath locators. In Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 3–6 November 2014; pp. 449–454.
40. Leotta, M.; Stocco, A.; Ricca, F.; Tonella, P. Using multi-locators to increase the robustness of web test cases. In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015; pp. 1–10.
41. Presler-Marshall, K.; Horton, E.; Heckman, S.; Stolee, K. Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness. In Proceedings of the 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), Montreal, QC, Canada, 27 May 2019; pp. 7–13.
42. Chhajed, S. *Learning ELK Stack*; Packt Publishing Ltd.: Birmingham, UK, 2015.
43. Bertolino, A.; Calabró, A.; De Angelis, G.; Gallego, M.; García, B.; Gortázar, F. When the testing gets tough, the tough get ElasTest. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May–3 June 2018; pp. 17–20.
44. Bradski, G.; Kaehler, A. *Learning OpenCV: Computer Vision with the OpenCV Library*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2008.
45. Downing, S.M. Validity: On the meaningful interpretation of assessment data. *Med. Educ.* **2003**, *37*, 830–837. [CrossRef] [PubMed]