Optical Character Recognition Based-On System for Automated Software Testing

D. Abbas and J. I. Olszewska
School of Computing and Engineering, University of the West of Scotland, U.K.

Keywords: Intelligent Systems, Autonomous Systems, Trustworthy Artificial Intelligence, Expert Systems, Software Robots, Automated Software Testing, Machine Learning, Optical Character Recognition, Computer Vision.

Abstract: The paper presents the development and deployment of an artificial intelligence (AI) test automation framework that allows testers to more fluidly develop scripts and carry out their day-to-day tasks. In particular, the framework aims to speed up the test automation process by enabling its users to locate elements on a webpage through the use of template-matching-based image recognition as well as optical character recognition (OCR). Indeed, test automation specialists spend much of their time creating page-object models (POMs), where they capture elements on the screen via complex locators such as cascading style sheet (CSS) or XPath. However, when webpages are updated or elements are moved around, locators become void, eventually pointing to nothing unless written in such a dynamic way as to prevent this. This heavily relies on developers providing meaningful tags to elements that they can then be located by, whereas with the introduction of an image recognition engine in our AI framework, this tedious and long-winded approach has been be shortened.

1 INTRODUCTION

As advancements are made in technology, the approaches and methodologies to accurately test such technologies must also evolve (Black et al., 2022). Indeed, reliable software testing is required to allow trustworthy autonomous systems, multi-agent systems, and/or robotic systems to evolve close by and/or interact with humans such as companion robots in assistive-living environments; autonomous vehicles in smart cities; or cloud robotics systems in smart manufacturing (Olszewska, 2020).

While the goal of testing is mainly to verify the quality, performance, or reliability of whatever is tested, testing in the modern age is a complex and nuanced field that is comprised of over a dozen of different types of testing (IEEE, 2021). These ones can be broadly split in *functional testing*, such as regression testing (Long, 1993) to catch a large class of bugs quickly and efficiently, and *non-functional testing* such as usability testing, and especially pattern-based usability testing (Dias and Paiva, 2017) to test usability guidelines (or best practices) by defining generic test strategies (i.e. test patterns) in order to allow testing usability aspects on web applications.

Even though manual testing is an integral part of the testing process and includes the development of a test strategy, test plan, test cases and test scripts (Alferidah and Ahmed, 2020), automated testing is required to cope with the pace of the software's continuously integrated/continuously developed (CI/CD) pipeline that streamlines the development and testing process within a software development life-cycle (SDLC) and aims to automatically fire off automated regression tests upon deployment (Chowdhury, A. R., 2023).

Whilst the idea of test automation is straightforward, the implementation and integration of automated software testing into an SDLC can be an intricate process.

So to overcome this aspect, a test automation framework provides rules, guidelines, and tools that the user can utilise to write test scripts (Celik et al., 2017). It can also be seen as a structure that provides an environment where automated test scripts can be executed (Chowdhury, A. R., 2023). Some of the major components of test automation frameworks usually consist of the test data management and testing libraries, including unit testing, integration testing and behaviour-driven development (Chowdhury, A. R., 2023). It is worth noting that there are many types of test automation frameworks (Chowdhury, A. R., 2023), e.g. modular testing-, data-driven testing-, keywords-driven testing-, hybrid testing-,

Abbas, D. and Olszewska, J.
Optical Character Recognition Based-On System for Automated Software Testing.
DOI: 10.5220/0012740000003690

In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 1*, pages 894-906
ISBN: 978-989-758-692-7; ISSN: 2184-4992

or behaviour-driven development (BDD) framework. Hence, test automation can ultimately be implemented in a variety of ways via different techniques, ranging from simple capture and replace and finishing with more sophisticated ones such as keyword- and process-driven approaches (Gafurov et al., 2018).

In particular, the behaviour-driven development (BDD) framework (Knight, A., 2017), which extends the test-driven development (TDD) approach (Sheshasaayee and Banumathi, 2018) by focusing on user requirements and expectations as well as enabling collaboration and automation, can be integrated into DevOps (Gohil et al., 2011) and further software quality verification (Cavalcante and Sales, 2018). For this purpose, BDD consists in writing requirements in a structured and testable format (i.e. *feature files* which are used to describe test scenario in a structured natural language (Yang et al., 2019) and *step definitions* which are abstractions representing the elements in a scenario such as contexts, events, and actions (Solis and Wang, 2011), with each step in the scenario associated with a corresponding step implementation function in the underlying programming language (Storer and Bob, 2019)) that can be evaluated to ensure compliance with the expected behavior (Farooq et al., 2023).

On the other hand, several tools have been developed for the test automation with old ones such as QuickTest Professional (QTP) (Wang and He, 2014) with keyword-driven methodology, which is a scripting technique that uses data files to contain the keyword related to the system under test (SUT) (Hamilton, T., 2023), used for the functional testing and regression testing (Lenka et al., 2018). Newer tools include the development of solutions like Selenium, which is an open-source, highly customisable, cross-Browser, web-testing automation framework (Ramya et al., 2017). The most recent tools are AI-assisted and AI-driven tools like Applitools (Calantonio, J., 2023) and may involve visual graphical user interface (GUI) testing capabilities (Alferidah and Ahmed, 2020).

It is worth noting that the decision to automate is the first step in the automation testing life cycle methodology (ATLM), and not all projects will require automation nor will meet the correct criteria for automation (Borjesson, 2012). Actually, test automation is the automation of tests that have already been run and verified to be working correctly. It is a step out of the typical software testing life cycle (STLC) (Hourani et al., 2019). So, test automation is carried out after the STLC has completed, and can only be considered if pre-requisites such as product owner (PO) interest and return on investment (ROI) on test automation have been achieved, because test automation requires a substantial initial investment and the benefits of the seeds of automation only begin to sprout after several weeks and months.

Therefore, test automation can go from no automation (i.e. manual testing), to automated testing and beyond, i.e. to self-managed, self-optimized testing, or even to autonomous testing such as self-testing, self-healing or self-repair (Eldh, 2020). For these latter stages of automation, using AI for software testing has a lot of potential and may improve quality assurance (Hourani et al., 2019).

In particular, AI-assisted test automation, which is also referred to as *AI-driven testing*, is concerned with the use of AI/ML technologies in the performance of automated testing activities (King et al., 2019), and implementing AI/ML can help among others to streamline the test automation process and offer functionalities such as test case creation models, making the software testing even more efficient and bugs easier to catch (Drugeot, C., 2020). Furthermore, machine learning (ML) classifiers can aid to predict defective software modules, most notably in safety-critical systems (Moreira Nascimento et al., 2019).

Moreover, the use of computer-vision-based techniques can lead to robotic process automation (RPA) (Yatskiv et al., 2020) or software robots for test automation (Gao et al., 2019).

Indeed, since automated software testing needs data (Zhu, 2018), and considering that 'the source code is data, and the screens, websites, databases, input and output are just data' (Hourani et al., 2019), machine learning and computer vision techniques such as *template matching* and *optical character recognition (OCR)* (Olszewska, 2015) can help collect data in the form of images for application screens (Yu et al., 2019) and widgets (Qian et al., 2023) and manage such data (Amershi et al., 2019) in context of test automation (King et al., 2019). In particular, template matching is a technique in digital image processing for finding small parts of an image which match a template image (Kalina and Golovanov, 2019), while OCR is a study of digital image processing for extracting alphanumeric data from images through pre-processing, segmentation, feature extraction, and recognition (Hananto et al., 2023).

Thence, testing with *visual GUI testing* (Borjesson, 2012), which is also known as *visually validation testing* (Borjesson and Feldt, 2012), is conducted with tool support that uses such image recognition algorithms and automated scripts to perform tests through GUI interaction. GUI interaction works on the highest level of system abstraction and allows the technique to emulate end-user behaviour to automate complex user

scenarios (Wheeler and Olszewska, 2022). User scenarios can therefore, with this technique, perceivably be executed faster, with higher frequency, at lower cost, with gained quality, etc. (Leotta et al., 2013).

So in this study, we utilise BDD techniques to organise test suites, develop readable test scenarios, and configure and run test runs. This is designed to work in tandem with the AI/ML aspect of the framework enhanced by computer-vision techniques for the OCR and template matching algorithms, in order to provide an automation engineer with a complete experience.

Hence, this work aims to deliver an AI-assisted test automation framework that leverages such technologies to enhance the test automation experience, and our framework goal being to minimise the effort of building a test automation framework and collecting data.

On the other hand, robust test execution is another goal the developed framework in this work has sought to achieve; the framework provides this level of test robustness using a two-fold approach. This mixes traditional as well as visual GUI testing elements to provide a more full-bodied experience capable of discovering bugs at the GUI level as well as at the document-object-model (DOM) level.

Besides, the AI-based automated testing framework has been developed following the D7-R4 approach (Olszewska, 2019).

The paper is structured as follows. In Section 2, we present the developed AI-based automated testing framework, while Section 3 compares productivity across various test automation scenarios most experienced by software testers. Conclusions are drawn up in Section 4.

2 PROPOSED APPROACH

This section covers the various technologies and software (Section 2.1) along with the testing methods and features (Sections 2.2-2.3) at play within the developed framework (see Fig. 1) as well as the setting up of the visual validation algorithms and testing (Sections 2.4-2.5).

2.1 Test Environment

In order to deliver the proposed framework (see Fig. 1) in a working capacity, our framework has been coded in Python (TechVidVan, 2023), using PyCharm integrated development environment (IDE), along with Python's built-in libraries and NumPy library supporting large, multi-dimensional arrays and matrices and a large collection of high-level mathematical functions to operate on these arrays. The development of our open-source system also involved the use of test suite core librairies such as PyTest, PyTest-BDD, and Selenium as well as a series of computer-vision librairies such as PyTesseract and OpenCV, as explained further in this section.

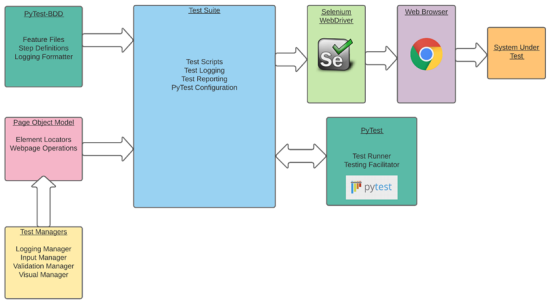

Figure 1: Developed framework architecture.

2.1.1 Selenium

Selenium is an open-source test automation library that utilises a WebDriver to interact with browsers in context of web-based application testing and that has seen widespread popularity within the industry in recent years (Tanaka et al., 2020). The library provides basic functions such as clicking on an element or selecting by index from a drop-down list among others, allowing testers to simulate common activities performed by end-users and to build regression packs (Selenium, 2022).

In this work, it has been specifically chosen for this reason, since it allows us to utilize the Selenium WebDriver, which is as the name suggests what will drive the tests. Indeed, the WebDriver communicates directly with a browser and uses its native support for automating the execution process of the test cases (Ramya et al., 2017). Much like Selenium itself, Selenium WebDriver can work on any browser given that the respective driver has been developed, with all major browsers being supported. It is worth adding that for this project, the main web browser that all the carried-out tests take place on is *Google Chrome* due to its popularity and widespread use.

2.1.2 PyTest

PyTest is a Python testing framework that can be used for various levels of testing, including unit tests, integration tests, end-to-end tests, and functional tests. Its features include parametrized testing, fixtures, and assert re-writing (Hunt, 2023).

In the case of our system, PyTest is primarily used for its fixtures which are used to instantiate and yield the various test managers that are used throughout the framework. The reasoning for this is so that the user

does not ever need to instantiate the managers themselves and that all of them will work in alongside the logging module to deliver seamless test logging and aid reporting.

PyTest is also used to run and organise the tests, which is done so via starting the function's name with 'test', this later integrates with PyTest-BDD to help us locate our feature file, containing our scenarios (tests); pull the steps from the scenario; locate the associated code; and run the code.

2.1.3 PyTest-BDD

PyTest-BDD is a PyTest add-on that implements a subset of the Gherkin language for automating project requirements testing and that enables behaviour driven development (BDD) within the framework (Santos et al., 2022). Hence, it allows an engineer to take a behaviour-centric approach to the development of their scripts.

This approach has been taken primarily for two reasons, namely, *complexity abstraction* (i.e. keeping the complex code that carries out the tasks separate from the steps of the test, enhancing readability and understanding of the system as a whole) and *advanced structuring* (i.e. structuring the test suite in such a way that complexity and code is modular and relevant, which aids compartmentalisation and understanding of the framework).

The BDD element of the framework enables the user to organise their tests via *feature files*, *step definitions*, and *page-object models* (POM) (i.e. helper files that are a way of implementing and abstracting the code that will run within the step definitions, by modeling the web pages involved in the test process as 'objects') (Leotta et al., 2013).

2.1.4 PyTesseract

PyTesseract is the Python library derived from Tesseract (Zelic, F. and Sable, A., 2023) which is a very well-known optical character recognition (OCR) tool (Bugayong et al., 2022).

Indeed, Tesseract is an open-source project that provides an OCR engine capable of advanced image recognition in a variety of formats (Smith, 2007). The steps in which Tesseract takes to optically recognize characters are, namely, *word finding*, *line finding*, and *character classification*. Word and line finding attempts to locate the rough areas of text, which are then organized into blobs (Zelic, F. and Sable, A., 2023). These blobs are then broken down into words and characters which the engine attempts to sequentially recognize. Having successfully recognized a word or set of words, this then further trains the model (Zelic, F. and Sable, A., 2023). Tesseract itself is capable of recognizing more than 100 languages and can be trained to recognize and interpret many more (Google Open Source, 2021).

The reason for its choice for our work specifically is that it is the most robust and popular OCR library available. Moreover, it is open-source, meaning its use and extension comes with no charge.

In the case of our framework, the text is read from the input image which is the screenshot of web elements that are provided by the Selenium WebDriver. Actually, the screenshot is first processed, then fed to the OCR engine, processed once more, and finally text is then output. It is precisely this text that is compared to the 'text' or 'value' attribute of that element.

2.1.5 OpenCV

OpenCV is a comprehensive and open-source computer vision and machine learning software library. It contains more than 2,500 optimized algorithms and is one of the most robust and popular options available on the market (Zelic, F. and Sable, A., 2023).

The library provides several methods and functionalities surrounding computer vision, which is the basis of our visual validation testing and facilitates a lot of the image handling, pre-processing, template matching, and post-processing required for visual validation testing.

In particular, OpenCV is used in the following ways within our framework to deliver visual validation testing:

- *Decoding images* - this is necessary as part of the pre-processing of images needed for template matching;
- *Colour conversion* - this is another step in pre-processing images for template matching;
- *Reading files* - OpenCV allows us to read images and store them as variables and objects;
- *Template matching* - this allows us to pinpoint the location of our template within a source image using various matching algorithms.

Hence, the use of OpenCV in our framework contributes to provide added security and robustness to the automated testing delivered by our system.

2.2 Test Managers

What the framework essentially provides an automation engineer is the necessary methods and tools from which they can choose and utilise to carry out their test automation efforts. However, to properly abstract complexity within the framework so that the process

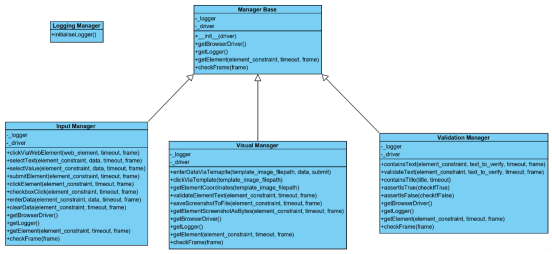

Figure 2: Test manager class diagram.

of utilising, it is necessary to develop the test managers.

The test managers work in harmony to provide the testers with what they might need; this is done by splitting various functional areas into the managers themselves. The test managers each deal with an integral area of test automation and were developed in the following order: (1) *manager base*, (2) *logging manager*, (3) *input manager*, (4) *validation manager*, and (5) *visual manager*, as detailed in the remaining part of this section.

The class diagram in Fig. 2 showcases the relationships between the test managers which are essentially classes revolving around a certain layer of test automation. The visual, input, and validation managers inherit from the manager base which gives them access to not only the parent methods but the driver and logger that are used throughout the testing process.

2.2.1 Manager Base

Manager Base acts as the base class for the test managers. This allows them to all have access to the same instance of the driver and logger that are both integral to the test automation. The Selenium WebDriver is what is used to create the browser instance upon which all the respective code is run, and the logger simply gives access to the active logger object to ensure the output remains consistent.

As well as providing access to the driver and logger objects, it provides the user with two basic methods, namely, *getElement*, which attempts to find and return the element that is passed in by using an XPath locator and *check frame*, which checks the current frame and whether or not we are in the correct frame before switching back to default. This is important for when we are working with iframes which are essentially windows inside windows that must be navigated to before work can be done on the elements inside.

2.2.2 Logging Manager

Logging Manager has no association whatsoever to any of the other test manager classes, as it can be seen in the class diagram displayed in Fig. 2. This is purely because the purpose of the logging manager class is to initialise the logger object and apply the necessary settings for the console output.

To properly output to the console on top of the logger's baseline capabilities, we need to set the formatting and logging levels. Thence, the *initialiseLogger()* method first creates the new logger object, followed by setting the level, the name and path of the eventual .log file as well as the formatting of the output. Once the logger has been initialised once, we can reference it by utilising *logging.getLogger(NameOfLogger)*. This references Python's internal logging library, which allows us to access the same instance of the logger throughout the project and various classes and files. The above method is in the manager base, which as aforementioned is used as the base class for all the test managers - excluding the logging manager. This is precisely how each of the managers can utilize the same logger object for logging and reporting.

2.2.3 Input Manager

Input Manager facilitates all the input commands that are used to interact with the web browser. These methods enable the user to interact with a webpage using the most common and popular types of interaction such as *clickViaWebElement*, *selectText*, *selectValue*, *submitElement*, *clickElement*, *checkboxClick*, *enterData*, and *clearData*. Each of these methods provided by the input manager enables the automation engineer to interact with the webpage. The purpose for abstracting these functions through a test manager is for primarily two reasons: to access to the same logging output and same driver. This essentially allows the engineer to write code without having to worry about reporting and logging, which are automatically taken care of as long as they use the provided functionality from the test managers.

So, of the three core test managers that will be used by the automation engineer (i.e. input, validation, visual), the input manager will undoubtedly see the most use.

2.2.4 Validation Manager

Validation Manager is undeniably the most important to the testing process, whilst not used as frequently as the input or validation managers. The validation manager essentially allows the engineer to validate various kinds of information and data, via assertions and other means, and is the primary way for the engineer to ensure robustness in functionality.

The validation manager enables the user to use

the following methods: *containsText* (which checks that the text within an element matches contains what we want it to, i.e. a word from a sentence or paragraph); *validateText* (which validates that the text within the element is a 1:1 match for what we are looking for); *containsTitle* (which checks that the title of the webpage - shown in the tab near the top - is as we expect); *assertIsTrue* (which asserts that something equals true); and *assertIsFalse* (which asserts that something equals to false). These five methods are what the validation manager has to offer an engineer that is using the framework. While it does provide the majority of the validation methods that may be used in web testing, it also lends itself to extension for further development and expansion, as do all the managers and the framework as a whole.

2.2.5 Visual Manager

Visual Manager, which as the name suggests, handles the visual validation testing and provides to the engineer operations such as optical character recognition (OCR) and template matching for image recognition-based testing. The purpose for this manager on top of the existing managers is to provide another layer of testing that can further confirm the robustness and integrity of the system under test (SUT). By doing this, we are better achieving the goal of automation which is to (ideally) be run upon new deployments to gain an understanding of the overall system status. The methods provided by this manager are as follows: *enterDataViaTemplate* (which uses a template image parameter to locate the area on the screen that we want to interact with, then clicks on this area if found and enters in the parameterized data); *clickViaTemplate* (which clicks at the x/y coordinates of our located template); *getElementCoordinates* (which returns a Numpy array with the x and y coordinates of the located template); *validateElementText* (which uses the PyTesseract library to implement OCR, takes a screenshot of the element and simultaneously grabs the text attribute of said element and compares them); *saveScreenshotToFile* (which saves a screenshot to file, most typically will be used for web element screenshot library); and *getElementScreenshotAsBytes* (which returns both the screenshot of a web element in byte form and the text of the web element and is used by other methods within the class).

The visual manager is the most complex of the managers and provides a breadth of visual validation testing in tandem with the other managers to aid in providing a well-rounded and robust testing experience.

2.3 Test Logging and Reporting

Test reporting and logging were implemented after the core test suite functionality was in place. This was primarily done to deliver a more coherent environment for an engineer to develop their tests in, as described below.

2.3.1 Test Logging

The logging of tests and the events that take place during a test was one of the first developments as part of the framework. Being able to keep track of what is happening during the test and afterwards is crucial, not only from a testing point of view but also from a debugging angle.

At every stage of development for a test script, it is re-run to ensure what has been developed is working, and to better facilitate this commonality among automation engineers, test logging was deemed an integral part of the framework. This allows for monitoring of the testing and debugging, but also for easily indicating whether a test has passed or failed and exactly at what step of the test.

The PyTest-BDD library allows for the configuration of the output using Gherkin syntax. This allows us to output formatted code, showcasing the scenario and steps that were executed, alongside whether the test as a whole passed or failed. The end product of the logging is the .log file, which includes all the logs that fired during the test run. Unlike reports which need to be generated manually, the .log files are automatically generated and easy to handle due to their generally small size.

2.3.2 Test Reporting

Reporting must be run manually via the command line. At first, a JavaScript Object Notation (JSON) object must be generated from the test run, and then the report itself can be generated from this. The decision to make this process manual is based on the fact that - of a hundred test runs, only maybe one report will be sent to the stakeholders - it is not something that needs to be constantly generated as it can quickly begin to take up a lot of disk space.

For this reason and to reduce post-processing times, the reporting was made to be a manual process via the command-line interface (CLI). The reporting itself is powered using PyTest-HTML (a built-in library that is coupled with PyTest) that allows us to provide basic reporting that gives an overview of what has passed and failed in a Hyper Text Mark-up Language (.html) format.

2.4 Template Matching

Template Matching is one of the two ways that visual validation testing is provided by the framework and is one of the latter areas that were developed once the core and aforementioned managers were implemented. The framework provides this functionality to the automation engineer by utilising several libraries in tandem.

Template matching essentially consists in finding an image within an image. Traditionally, template matching has been used in image recognition applications. Our framework utilises template matching by detecting a template image within a source image. In the case of this work, the source image is a screenshot of the webpage that is taken using the WebDriver; this source image being the basis of the template matching algorithm. Soon after, a template - or image we want to find within the source image - must be defined. Once these two key components have been defined and pre-processed, we can execute the template matching.

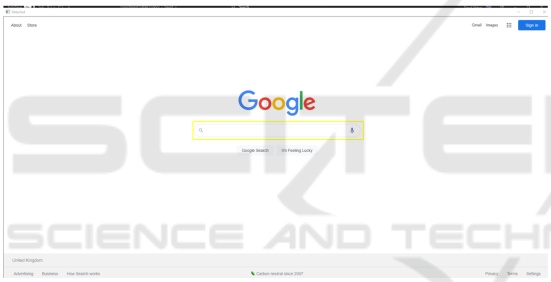

Figure 3: Matched template example.

An example of template matching in action can be seen in Fig. 3, which showcases the Google homepage search bar being found by the template matching algorithm. For demonstration, the detected area is surrounded by a yellow rectangle when utilising OpenCV. Indeed, this entire window is generated using OpenCV, which opens a prompt window showcasing the matched region. In this scenario, the template is the search bar element screenshot, whilst the source image is the entirety of the Google homepage.

So the process flow for the *getElementCoordinates* method within the visual manager class can be described as follows. A screenshot of the webpage is taken using the Selenium WebDriver - this acts as the source image. Then, the source image is decoded and converted to grayscale (as part of the pre-processing). Next, the template is read from the file path provided by the engineer and parameterized as part of the *getElementCoordinates* method. Then, the width and height of the template are stored as variables. Next, using OpenCV, we run the template matching algorithm and capture the result which is then converted into a one-dimensional array using NumPy. Next, the one-dimensional array is unravelled and converted into a multi-dimensional NumPy array - this is what is returned from the *getElementCoordinates* method, the multi-dimensional array holds the x and y coordinates that are passed to the Selenium ActionChains library later on to offset the cursor to the correct x and y coordinates on the screen to interact with the webpage. Next, the multi-dimensional array (i.e. the x and y coordinates) are returned from the method. Finally, once the x and y coordinates are returned from the *getElementCoordinates* method, we click on the centre of the matched region by multiplying the width and height of the template by 0.5 (i.e. template width $\times$ 0.5). Now that we have interacted with the element and brought it to focus, we can continue with whatever operations the automation engineer would like to do such as entering data.

2.5 Optical Character Recognition

Optical Character Recognition (OCR) is the second way in which visual validation testing is provided by the framework. This is done by utilising PyTesseract which allows the use of the Tesseract.exe image recognition engine. The path to this .exe will differ depending on the machine and must be updated in the OCR method called *validateElementText*.

OCR sees widespread use in a variety of applications in the modern-day, such as automatic number plate recognition, QR code scanning, language translation (Kalina and Golovanov, 2019) to name a few. It is an integral aspect of many computer-vision-related applications and software. In its purest form, OCR is a subset of pattern recognition problems, which forms its basis from several processes including but not limited to input data pre-processing, segmentation, and feature extraction (Kalina and Golovanov, 2019).

Furthermore, the use of OCR within the test automation framework is a crucial step in implementing visual validation testing on top of more traditional methods, where elements are accessed at the document object model (DOM) level. This approach, when coupled with traditional test automation methods provides a two-fold layer of robustness, where we are not only verifying the contents of the element at the DOM level but also from a visual perspective.

The main benefit of this is that it is closer to how a human would interact with the application, ergo making it more realistic. When manually testing, testers will not inspect elements to ensure that the values, tags, and attributes are correct - it is a test that relies heavily on the tester's vision. Thus, by utilising

both methods of test automation, the test automation framework provides greater levels of confidence to an engineer, as well as test reliability and test integrity. This in turn gives the business confidence in what they have developed and in certain cases, can even boost morale and productivity within the team.

```
# Using tesseract we employ OCR to validate element text is equal to what we wish it to be
def validateElementText(self, element_constraint, timeout=5, frame="default"):
    screenshot_bytes, element_text = self.getElementScreenshotAsBytes(element_constraint, timeout, frame)
    decoded_img = cv2.imdecode(numpy.frombuffer(screenshot_bytes, numpy.uint8), -1)

    pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'
    text_from_screenshot = pytesseract.image_to_string(decoded_img)

    # Here we manipulate the string to remove the trailing characters
    text_from_screenshot = "\n".join([ll.rstrip() for ll in text_from_screenshot.splitlines() if ll.strip()])
    self._logger.info("Text From Element Screenshot Found: " + text_from_screenshot)

    try:
        assert text_from_screenshot == element_text
        self._logger.info("Success: Text Matches On-Screen")

    except Exception:
        self._logger.exception("Failure: Text Does Not Match On-Screen")
        raise
```

Figure 4: validateElementText method utilising OCR engine.

Figure 4 is an extract from the code behind the framework, that employs the use of OCR to validate the text of an element. At first, it utilizes the Selenium WebDriver to capture two variables, namely, *screenshot_bytes* (this is a screenshot of the web element taken by using the WebDriver, which is later fed to the OCR engine) and *element_text* (this is the string from the text attribute of the element that is passed in). Next, it uses the OpenCV library to decode the image so that it can run the OCR engine on it to capture the text. Once it has captured the text, it manipulates the string to get only the text we are interested in - this is a solution to an issue with PyTesseract where it occasionally adds trailing spaces or other special characters. Finally, it asserts that the text held in the element's text attribute is equal to the string that has been captured from the screenshot of the web element. So this is how OCR has been implemented within the framework, with its main aim being to provide a heightened level of robustness to the testing process.

3 APPLICATION AND DISCUSSION

In this section, we will describe the application of our AI-assisted automated testing framework in real-world context (in Section 3.1) and its evaluation (in Section 3.2) in terms of processing time and success rate of the computer vision algorithms which are embedded in our framework.

3.1 Application

Our developed framework for automated testing can aid the test building process in three domains, as follows: (i) writing the feature file (test scenario); (ii) coding the page-object model (POM), and (iii) attaching the code via the step definition, as explained in the remaining part of this section.

3.1.1 Writing the Feature File

The feature file is what holds one to many test scenarios, which are our tests. A feature file is written in Gherkin syntax, which is practically identical to the English language.

As a running example shown in Fig. 5, we create a .feature file under the 'features' module with a descriptive name. Now that the .feature file is created, we have to name the feature. The name of the feature should be identical to the name of the file for ease of understanding, as highlighted in red in Fig. 5.

Then, we should give the feature a description - this can be in plain English and acts as a way for any user that reads this feature file to get an understanding of what kind of tests are within the file.

Once the name of the feature file and a description are established, we can begin to write a test scenario, starting by naming the scenario something descriptive that makes sense to a human tester.

Next, when writing a test scenario in Gherkin syntax, we must prefix each step with the following operators 'Given', 'When' or 'Then'. These serve no real purpose besides enhanced readability, but a common rule is to use 'Given' steps as the setup for the rest of the test; 'When' steps as actions (e.g., 'When I search for Amazon'); and 'Then' steps as verifications or assertions.

3.1.2 Coding the Page-Object Model

The Page-Object Model (POM) is where all the coding within the framework is done primarily. Each POM will refer to a webpage (i.e. Google Homepage), and on each website, there will be various actions one can do such as search, select, enter, etc. - these will be the methods.

To continue with the running example, a .py file should be created under the *page_object_models* module in the project, naming it in a relevant way to the intended test. It is worth noting that if the test interacts with multiple websites/webpages, there is a need for many POMs. Then, within the POM, we should start by implementing the imports to access the methods made available by the test managers.

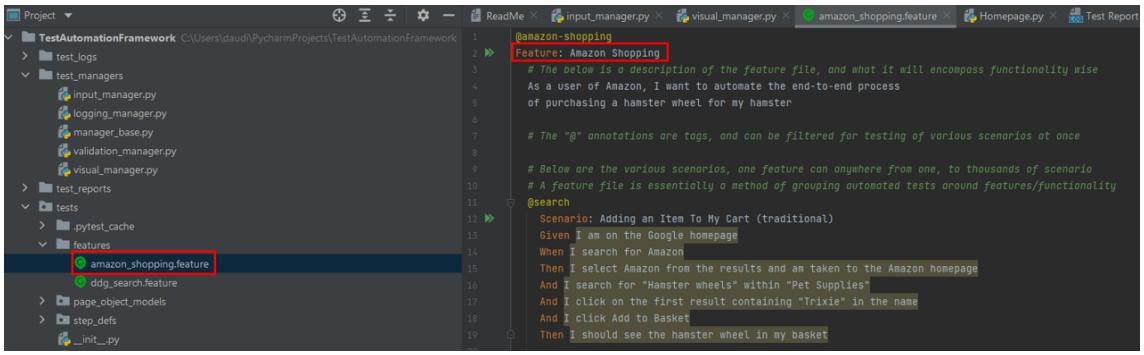

Figure 5: Example of 'Writing the Feature File' operation.

Once we have the POM class, it is best practice to capture locators and store them as private variables at the top of the class, as illustrated in Fig. 6.

```
class amazonHelper(object):
    # Protected attributes
    _driver: WebDriver
    _inputManager: inputManager
    _validationManager: validationManager
    _visualManager: visualManager

    # Locators (private)
    __search_category_select = "//select[@id = 'searchDropdownBox']"
    __search_bar_input = "//input[@id = 'twotabsearchtextbox']"
    __add_to_basket_btn = "//input[@value = 'Add to Basket']"
    __basket_link = "//a[@id = 'nav-cart']"
    __accept_cookie_policy_btn = "//input[@id='sp-cc-accept']"

    # Constructor
    def __init__(self, driver):
        # Class Attributes
        self._driver = driver
        self._inputManager = inputManager(driver)
        self._validationManager = validationManager(driver)
        self._visualManager = visualManager(driver)
```

Figure 6: Example of locators.

It is worth noting that any managers that one wishes to use also need to be referenced, ideally as protected attributes.

Next, methods (i.e. operations) for the respective webpage/website can be written, as shown in Fig. 7.

```
# Methods
# Accepts the cookie policy popup
def acceptCookiePolicy(self, visual=False):
    if visual is True:
        self._visualManager.clickViaTemplate(
            r"..\..\web_element_screenshots\amazon_accept_cookies_btn.png")
    else:
        self._inputManager.submitElement(self.__accept_cookie_policy_btn)

# Selects the correct category, inputs the search term and searches for the item
def search(self, category, search_term, visual=False):
    if visual is True:
        self._validationManager.containsTitle("Amazon")
        self._visualManager.enterDataViaTemplate(
            r"..\..\web_element_screenshots\amazon_category_selection.png", category, True)
        self._visualManager.enterDataViaTemplate(
            r"..\..\web_element_screenshots\amazon_searchbar_input.png", search_term, True)
        self._validationManager.containsTitle("Amazon.co.uk : " + search_term)
    else:
        self._validationManager.containsTitle("Amazon")
        self._inputManager.selectText(self.__search_category_select, category)
        self._inputManager.enterData(self.__search_bar_input, search_term)
        self._inputManager.getElement(self.__search_bar_input).send_keys(Keys.ENTER)
        self._validationManager.containsTitle("Amazon.co.uk : " + search_term)
```

Figure 7: Example of methods.

We can create as many or as few methods as we like, as long as all the functionality necessary to cover the test scenario that we wrote earlier has been fulfilled.

3.1.3 Attaching the Code via the Step Definition

Now that we have written the feature file as well as the POM, we are ready to 'glue the code' together to run our test.

For that purpose, under the *step_defs* module, we create a .py file with a name that is almost identical to the name of our scenario - this allows to keep track of what we have worked on. After creating such .py file, under the same module *step_defs*, the imports are copied and pasted in the 'Imports' text into the new step definitions file.

Now that we have the step definition file setup, we can generate the step definition code via a command using the terminal, as follows (see Fig. 8).

```
pytest-bdd generate features/some.feature > tests/functional/test_some.py
```

Figure 8: Example of step definition template code generation.

Next, at the top of the step definition file above all the generated methods, we can write a method using the @scenario decorator, with the path to our feature file and the name of our scenario, as displayed in Fig. 9. When we prefix the method with 'test', this allows PyTest to find it.

```
@scenario('..\\features\\amazon_shopping.feature', 'Adding an Item To My Cart (traditional)')
def test_adding_an_item_to_my_cart():
    pass
```

Figure 9: Example of method using the @scenario decorator.

Then, we add in all the respective methods calls for our operations and in the conftest.py file, we add a method that will yield the class object so that one can use the respective methods, as illustrated in Figs 10 and 11, respectively.

Finally, we can right-click on the step definition file and run the test by clinking 'Run PyTest'.

```python
@when('I search for Amazon')
def i_search_for_amazon(_googleHelper):
    _googleHelper.search("Amazon")


@then('I click Add to Basket')
def i_click_add_to_basket(_amazonHelper):
    _amazonHelper.clickAddToBasketButton()


@then(parsers.parse('I click on the first result containing "{search_term}" in the name'))
def i_click_on_the_first_result(_amazonHelper, search_term):
    _amazonHelper.clickResultContainingSearchTerm(search_term, 0)


@then(parsers.parse('I search for "{search_term}" within "{category}"'))
def i_search_for_hamster_wheels(_amazonHelper, search_term, category):
    _amazonHelper.search(category, search_term)


@then('I select Amazon from the results and am taken to the Amazon homepage')
def i_select_amazon_from_the_results_and_am_taken_to_the_amazon_homepage(_googleHelper):
    _googleHelper.clickResultContainingSearchTerm("Amazon", 0)


@then('I should see the hamster wheel in my basket')
def i_should_see_the_hamster_wheel_in_my_basket(_amazonHelper):
    _amazonHelper.navigateToBasket()
    _amazonHelper.verifyItemInBasket("Trixie")
```

Figure 10: Example of operation calls.

```python
@pytest.fixture()
def _amazonHelper(driver):
    _amazonHelper = amazonHelper(driver)
    yield _amazonHelper
```

Figure 11: Example of method that yields the class object.

3.2 Discussion

This section look at quantifying and evaluating the performance of the computer vision algorithms embedded in our developed framework. Thus, the computational speed of these algorithms is quantified using the logging from the system output that is generated through the test managers.

3.2.1 Template Matching Evaluation

As aforementioned in Section 2.4, template matching is one of two ways that visual validation testing aided by AI/ML has been implemented into the framework.

Traditional methods although more time-consuming in their setup are typically faster as they access the DOM layer of the webpage. Whereas visual validation methods such as OCR and template matching take more time as the algorithm needs to work to match the region, it does this pixel by pixel until a satisfactory match is found in the case of template matching.

Due to how the algorithm works by starting at x(0) and y(0), it will take longer or shorter depending on the position of the element, because of this, the computational time in order to match the element can vary heavily. These variations in the processing time can be seen in Fig. 12 for the running example.

In the first instance where the *clickAtCoordinates* method is called (which utilises template matching), we can see in Fig. 12 that it takes approximately 7 seconds for this to take place, whilst the second instance takes less than 1 second. Both instances start

```
15:49:48,628 - INFO - Logging Handler Created
15:49:48,629 - INFO - Feature loaded: Amazon Shopping
15:49:48,629 - INFO - Scenario loaded: Adding an Item To My Cart (visual)
15:49:55,617 - INFO - Clicking Element at x[1027.] y[790.] Within Function {clickAtCoordinates}
15:50:04,121 - INFO - Clicking Element at x[963.] y[398.5] Within Function {clickAtCoordinates}
15:50:04,740 - INFO - Entering data Amazon Within Function {enterDataAtCoordinates}
15:50:04,779 - INFO - Retrieving Element {//input[@title = 'Search']} Within Function {getElement}
15:50:04,791 - INFO - Found Element {//input[@title = 'Search']} Within Function {getElement}
```

Figure 12: Sample of template matching processing time measures.

from the x and y coordinates 0, however the processing time varies depending on how far away the first element is from the starting point.

3.2.2 Optical Character Recognition Evaluation

As outlined in Section 2.5, OCR within the dimension of our framework provides a two-fold approach to error detection in the form of a dual check. This functionality is aimed at ensuring that tests are reliable and that if a step passes, there are no unforeseen issues.

Traditionally, one interact with the webpage via the DOM layer, however in contrast to this, normal users and manual testers will not test the system under test (SUT) like this. This means that there needs to be a way for the visual element of a webpage to be verified alongside the DOM layer, and that is exactly what OCR within our framework provides. This improves not only error detection, but results in finding errors sooner rather than later, all while using a more realistic approach to that of a normal user.

Due to the nature of the characters we are working with, in that they are clearly typed words and sentences, the OCR engine works very quickly, taking only a couple of hundred milliseconds in order to be completed. Again, using the logging output from tests that have run, we can observe in Fig. 13 that the OCR takes 310ms for the running example.

```
17:28:28,313 - INFO - Logging Handler Created
17:28:28,314 - INFO - Feature loaded: Amazon Shopping
17:28:28,314 - INFO - Scenario loaded: Adding an Item To My Cart (visual)
17:28:34,500 - INFO - Clicking Element at x[1205.] y[705.] Within Function {clickAtCoordinates}
17:29:03,062 - INFO - Clicking Element at x[963.] y[398.5] Within Function {clickAtCoordinates}
17:29:03,617 - INFO - Entering data Amazon Within Function {enterDataAtCoordinates}
17:29:03,640 - INFO - Retrieving Element {//input[@title = 'Search']} Within Function {getElement}
17:29:03,648 - INFO - Found Element {//input[@title = 'Search']} Within Function {getElement}
17:29:10,088 - INFO - Clicking Element at x[444.] y[496.] Within Function {clickAtCoordinates}
17:29:13,555 - INFO - Page Title: Amazon.co.uk: Low Prices in Electronics, Books, Sports Equipment & more
17:29:17,137 - INFO - Clicking Element at x[389.] y[29.] Within Function {clickAtCoordinates}
17:29:17,701 - INFO - Entering data Pet Supplies Within Function {enterDataAtCoordinates}
17:29:21,426 - INFO - Clicking Element at x[937.5] y[30.5] Within Function {clickAtCoordinates}
17:29:21,968 - INFO - Entering data Hamster wheels Within Function {enterDataAtCoordinates}
17:29:23,184 - INFO - Page Title: Amazon.co.uk : Hamster wheels
17:29:28,426 - INFO - Clicking Element at x[1422.] y[816.5] Within Function {clickAtCoordinates}
17:29:32,728 - INFO - Clicking Element at x[939.5] y[216.] Within Function {clickAtCoordinates}
17:29:39,410 - INFO - Clicking Element at x[1564.] y[506.5] Within Function {clickAtCoordinates}
17:29:47,351 - INFO - Clicking Element at x[1519.] y[117.] Within Function {clickAtCoordinates}
17:30:28,355 - INFO - Retrieving Element Screenshot {//span[contains(@class, 'product-title') and contains(text()
17:30:28,865 - INFO - Text From Element Screenshot Found: Pet        OCR Output
```

Figure 13: Sample of optical character recognition (OCR) processing time measures.

Overall, from this we can see that the computational speed for the OCR is reasonably fast. OCR is also reliable, since for all the instances where it was used, all 100% of those instances successfully recognized the characters from the screenshot of the web element. The reason that the OCR algorithm within

the framework is so reliable is because the characters are easy enough for the engine to recognise. This in comparison to attempting to recognise text from a news paper or handwriting means the accuracy of the optical character recognition in our AI-assisted automated testing framework is working flawlessly across the board.

4 CONCLUSIONS

This work has successfully developed an AI-assisted test automation framework that has shone a light on the potential of artificial intelligence (AI), machine learning (ML), and computer vision (CV) within the software testing industry, specifically automation.

Indeed, our AI-assisted test automation framework, that leverages visual and traditional testing methods, minimises the effort of automating tests and collecting data. Moreover, the use of computer vision techniques provide a two-fold layer that adds security and robustness to the automated testing delivered by our framework, with the contents of the element being not only verified at the DOM level but also from a visual perspective. Therefore, this AI-assisted test automation framework with embedded computer-vision capabilities and in tandem with BDD offers a complete automated software testing solution, usable for reliable testing in mission-critical applications.

REFERENCES

Alferidah, S. K. and Ahmed, S. (2020). Automated software testing tools. In *Proceedings of the IEEE International Conference on Computing and Information Technology*, pages 1–4.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 291–300.

Black, R., Davenport, J. H., Olszewska, J. I., Roessler, J., Smith, A. L., and Wright, J. (2022). *Artificial Intelligence and Software Testing: Building systems you can trust*. BCS Press.

Borjesson, E. (2012). Industrial applicability of visual GUI testing for system and acceptance test automation. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 475–478.

Borjesson, E. and Feldt, R. (2012). Automated system testing using visual GUI testing tools: A comparative study in industry. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 350–359.

Bugayong, V. E., Flores Villaverde, J., and Linsangan, N. B. (2022). Google tesseract: Optical character recognition (ocr) on hdd / ssd labels using machine vision. In *Proceedings of the IEEE International Conference on Computer and Automation Engineering*, pages 56–60.

Calantonio, J. (2023). 7 Innovative AI Test Automation Tools. Available at: https://testguild.com/7-innovative-ai-test-automation-tools-future-third-wave/.

Cavalcante, M. G. and Sales, J. I. (2018). The behavior driven development applied to the software quality test. In *Proceedings of the IEEE Iberian Conference on Information Systems and Technologies*, pages 1–4.

Celik, E., Eren, S., Cini, E., and Keles, O. (2017). Software test automation and a sample practice for an enterprise business software. In *Proceedings of the IEEE International Conference on Computer Science and Engineering*, pages 141–144.

Chowdhury, A. R. (2023). Testim. Your Complete Guide to Test Automation Frameworks. Available at: https://www.testim.io/blog/test-automation-frameworks/.

Dias, F. and Paiva, A. C. R. (2017). Pattern-based usability testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 366–371.

Drugeot, C. (2020). Software Testing News. How is AI Transforming Software Testing?

Eldh, S. (2020). Test automation improvement model - TAIM 2.0. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 334–337.

Farooq, M. S., Omer, U., Ramzan, A., Rasheed, M. A., and Atal, Z. (2023). Behavior driven development: A systematic literature review. *IEEE Access*, 11:88008–88024.

Gafurov, D., Hurum, A. E., and Markman, M. (2018). Achieving test automation with testers without coding skills: An industrial report. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 749–756.

Gao, J., Tao, C., Jie, D., and Lu, S. (2019). What is ai software testing? and why. In *Proceedings of the IEEE International Conference on Service-Oriented System Engineering*, pages 1–9.

Gohil, K., Alapati, N., and Joglekar, S. (2011). Towards behavior driven operations (bdops). In *Proceedings of the IEEE International Conference on Advances in Recent Technologies in Communication and Computing*, pages 262–264.

Google Open Source (2021). Tesseract OCR. Available at: https://github.com/tesseract-ocr/tesseract.

Hamilton, T. (2023). Keyword Driven Testing Framework with Example. Available at: https://www.guru99.com/keyword-driven-testing.html.

Hananto, A., Abdul Rahman, T. K., Brotosaputro, G., Fauzi, A., Hananto, A. L., and Priyatna, B. (2023). Parameters monitoring automation kiln manufacture based optical character recognition (OCR) with the template matching method. *International Journal of Intelligent*

Systems and Applications in Engineering, 11(6):621–635.

Hourani, H., Hammad, A., and Lafi, M. (2019). The impact of artificial intelligence on software testing. In *Proceedings of the IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology*, pages 565–570.

Hunt, J. (2023). PyTest Testing Framework. In *Advanced Guide to Python 3 Programming*. Springer.

IEEE (2021). IEEE/ISO/IEC 29119-4-2021 - International Standard - Software and systems engineering–Software testing–Part 4: Test techniques.

Kalina, D. and Golovanov, R. (2019). Application of template matching for optical character recognition. In *Proceedings of the IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering*, pages 2213–2217.

King, T. M., Arbon, J., Santiago, D., Adamo, D., Chin, W., and Shanmugam, R. (2019). Ai for testing today and tomorrow: Industry perspectives. In *Proceedings of the IEEE International Conference on Artificial Intelligence Testing*, pages 81–88.

Knight, A. (2017). 12 Awesome Benefits of BDD. Available at: https://automationpanda.com/2017/02/13/12-awesome-benefits-of-bdd/.

Lenka, R. K., Nayak, K. M., and Padhi, S. (2018). Automated testing tool: QTP. In *Proceedings of the IEEE International Conference on Advances in Computing, Communication Control and Networking*, pages 526–532.

Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. (2013). Repairing Selenium test cases: An industrial case study about web page element localization. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 487–488.

Long, M. A. (1993). Software regression testing success story. In *Proceedings of the IEEE International Test Conference*, pages 271–272.

Moreira Nascimento, A., Vismari, L. F., Cugnasca, P. S., Camargo Jr, J. B., and Rady de Almeira Jr, J. (2019). A cost-sensitive approach to enhance the use of ML classifiers in software testing efforts. In *Proceedings of the IEEE International Conference on Machine Learning and Applications*, pages 1806–1813.

Olszewska, J. I. (2015). Active contour based optical character recognition for automated scene understanding. *Neurocomputing*, 161(C):65–71.

Olszewska, J. I. (2019). D7-R4: Software development lifecycle for intelligent vision systems. In *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (KEOD)*, pages 435–441.

Olszewska, J. I. (2020). AI-T: Software testing ontology for AI-based systems. In *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (KEOD)*, pages 291–298.

Qian, J., Ma, Y., Lin, C., and Chen, L. (2023). Accelerating OCR-Based widget localization for test automation of GUI applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13.

Ramya, P., Sindhura, V., and Sagar, P. V. (2017). Testing using Selenium web driver. In *Proceedings of the IEEE International Conference on Electrical, Computer and Communication Technologies*, pages 1–7.

Santos, M. G. D., Petrillo, F., Halle, S., and Gueheneuc, Y.-G. (2022). An approach to apply automated acceptance testing for industrial robotic systems. In *Proceedings of the IEEE International Conference on Robotic Computing*, pages 336–337.

Selenium (2022). The Selenium project and tools. Available at: https://www.selenium.dev/documentation/en/introduction/the_selenium_project_and_tools/.

Sheshasaayee, A. and Banumathi, P. (2018). Impacts of behavioral driven development in the improvement of quality software deliverables. In *Proceedings of the IEEE International Conference on Inventive Computation Technologies*, pages 228–230.

Smith, R. (2007). An overview of the Tesseract OCR engine. In *Proceedings of the IEEE International Conference on Document Analysis and Recognition (ICDAR)*, pages 629–633.

Solis, C. and Wang, X. (2011). A study of the characteristics of behaviour driven development. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387.

Storer, T. and Bob, R. (2019). Behave nicely! automatic generation of code for behaviour driven development test suites. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 228–237.

Tanaka, T., Niibori, H., Shiyingxue, L., Nomura, S., Nakao, T., and Tsuda, K. (2020). Selenium based testing systems for analytical data generation of website user behavior. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 216–221.

TechVidVan (2023). Python Advantages and Disadvantages - Step in the right direction. Available at: https://techvidvan.com/tutorials/python-advantages-and-disadvantages/.

Wang, X. and He, G. (2014). The research of data-driven testing based on QTP. In *Proceedings of the IEEE Iberian Conference on Computer Science and Education*, pages 1063–1066.

Wheeler, D. and Olszewska, J. I. (2022). Cross-platform mobile application development for smart services. In *Proceedings of the IEEE Joint 22nd International Symposium on Computational Intelligence and Informatics and 8th International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics*, pages 203–208.

Yang, A. Z. H., Alencar da Costa, D., and Zou, Y. (2019). Predicting co-changes between functionality specifications and source code in behavior driven development. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories*, pages 534–544.

Yatskiv, N., Yatskiv, S., and Vasylyk, A. (2020). Method of robotic process automation in software testing using artificial intelligence. In *Proceedings of the IEEE International Conference on Advanced Computer Information Technologies*, pages 501–504.

Yu, S., Fang, C., Feng, Y., Zhao, W., and Chen, Z. (2019). Lirat: Layout and image recognition driving automated mobile testing of cross-platform. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 1066–1069.

Zelic, F. and Sable, A. (2023). OCR Unlocked: A Guide to Tesseract in Python with Pytesseract and OpenCV. Available at: https://nanonets.com/blog/ocr-with-tesseract/#technologyhowitworks#.

Zhu, H. (2018). Software testing as a problem of machine learning: Towards a foundation on computational learning theory. In *Proceedings of the IEEE/ACM International Workshop on Automation of Software Test*, pages 1–1.