Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ПОБУДОВА СИСТЕМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ ІНСТРУМЕНТІВ ШТУЧНОГО ІНТЕЛЕКТУ

Текстова частина
магістерської роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник магістерської роботи
д.т.н., доц. А. М. Глибовець

_____
(підпис)

“____” _____ 2021 р.

Виконав студент С. О. Сосницький
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики
Зав.кафедри інформатики,
к.ф.-м.н.
_____С. С. Гороховський
"____" _____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія програмного забезпечення Сосницькому Сергію Олександровичу

Тема: Побудова системи автоматизованого тестування з використання інструментів штучного інтелекту

Зміст текстової частини до магістерської роботи:
Зміст;
Анотація;
Вступ;

1. Дослідження галузі та аналіз проблем автоматизованого тестування;
2. Проектування системи автоматизованого тестування з використання інструментів штучного інтелекту;
3. Розробка системи;
4. Огляд результатів;

Висновки;
Список літератури;
Додатки.

Дата видачі "___" _____ 2021 р.
Керівник _____
Завдання отримав _____

Work schedule:

№	Phase name	Deadline	Note
1.	Obtaining the theme of the coursework.	01.11.2020	
2.	Thematic materials researching and collection.	15.11.2020	
3.	Work structure and goals discussion.	01.12.2020	
4.	Analysis of modern solutions and trends in the coursework area.	24.12.2020	
5.	Theoretical part: 1. Automated testing issues and challenges analysis; 2. Review and analysis of existing solution with AI components; 3. System design and architecture;	18.01.2021	
6.	The practical part implementation.	18.03.2021	
7.	Analysis and documentation of the practical part: 1. Data analysis and documentation; 2. Solution design and development; 3. Description system design and architecture; 4. Testing.	05.04.2021	
8.	Conclusion.	08.05.2021	
9.	Work revision.	09.05.2021	
10.	Corrections according to the mentor's remarks.	17.05.2020	
11.	Finalization of presentation materials.	20.05.2021	
12.	Presentation.	15.06.2020	

Student Sosnytskyi S. O.

Supervisor Hlybovets A.M.

“____” _____

Table of Contents

Abstract

In this paper, the author overviewed an automated testing application during software development, its issues, and challenges. Designed architecture and implemented a prototype for a system with a machine learning testing tool. A proposed design could resolve existing issues with user graphical interfaces tasting (GUI) by traditional function tools, and improve transparency and feedback to product managers and designers by verification user interfaces to original requirements and not only developed cases by the development team.

The system architecture design includes diagrams designed by the C4 + 1 model. As prototype components, TensorFlow and Keras libraries were used for machine learning tools, Cypress was a functional automation tool. Python and JavaScript as programming languages.

Keywords: automated testing, visual testing, graphical interface testing, machine learning, snapshot testing, continuous delivery, continuous code analysis.

Introduction

In our modern world, we get used to web applications, their convenience, simplicity, speed, and reliability. With the wide availability of internet connection options through different channels like wire, mobile, satellite, more and more customers change their habits in favor of online services. Because of this, more and more businesses are considering online as the main business platform to serve their customers everywhere across the globe. Every organization would like to deliver its software product or just a single feature in the tightest and previously not imaginable term in a matter of hours or even minutes. This trend pushes development teams to automate the software development phase where possible including software testing.

Software testing is an activity performed to assess and improve the quality of software. Automated software testing is the process where the testing functions such as test steps, launching, initialization, execution, analysis, and results output, are done automatically with the help of test automation tools.

Test automation as a practice started to be used in some companies quite a long time ago, however, it has become essential in recent years. The reason for this is the transition from the traditional Waterfall approach in software development to the Agile approach. In the traditional Waterfall approach you consequently plan, design, build the solution, then test the end product and fix issues that arise. It means that the end product must be tested only once during the corresponding phase in the software development cycle. If some tests fail, then one more test round is done after the issues have been fixed. So it is more cost-effective and makes more sense to test the product manually.

Currently, software projects are so sophisticated that it is not feasible to plan, design, and think overall the technical details in advance. The projects are usually long-term, while technologies and business needs change fast. It has become

extremely important for software products to respond to customers' feedback fast. So instead of having big rare releases, software companies have to have rapid delivery cycles and release new versions of software as quickly as possible.

Consequently, test automation is not a tool for developers anymore. Contrariwise it services and influences almost all stakeholders in software development organizations, including testers, developers, product owners, solution architects, DevOps specialists, project managers. Each organization has its own unique quality needs and demands for automation techniques and practices. The maturity of automation testing projects in some cases could lead to the failure of the whole project or elevation quality and customer satisfaction to new heights if testing done properly.

Machine learning (ML) in test automation is a completely new area, it could boost the productivity of test automation tools many times, improve software products quality and positively affect the whole organization. Several automation tools that provide machine learning capabilities have emerged in recent years, but they are still expensive niche products that are used in specific wide cases. Generic ML-based automation approach and tool that will service the development team and product managers yet to be discovered and created.

What makes this paper relevant, as were described above, automation testing is an obligatory component of any modern software development. Despite massive improvements in this sphere during the last decade, automation testing still remains a sophisticated solution that requires significant development efforts and in most cases still needs to be backed up by manual testing. Coupling automated testing with machine learning tools could boost quality and cooperation with stakeholders to a new level.

The objective of this paper is to discover existing solutions in the software testing area and in test automation specifically. Analyze and explore automated testing issues and benefits that could bring the usage of machine learning tools in this

area. Design the system architecture and implement a prototype to automatically compare website user interface with user interface design templates and make decisions if results are met requirements.

The object of study in this paper is automated testing with the use of elements of machine learning.

The research methods include discovery and research of existing solutions, articles, and papers. Discovery of the ongoing development of automated tools based on machine learning tools.

1. Industry research and analysis of automated testing issues and challenges

1.1 Automated Software Testing Overview

Software testing is usually understood as an activity performed to assess and improve the quality of software. In general, testing is based on the detection of defects and problems in software systems.

Automated software testing is the process where the main functions and test steps, such as launching, initialization, execution, analysis, and results output, are done automatically with the help of test automation tools.

Looking at the history of the test automation practices, we can see that some companies used test automation quite a long time ago, however, it has become essential in recent years. The reason for this is the transition from the traditional Waterfall approach in software development to the Agile approach. In the traditional Waterfall approach you consequentially plan, design, build the solution, then test the end product and fix issues that arise. It means that the end product must be tested only once during the corresponding phase in the software development cycle. If some tests fail, then one more test round is done after the issues have been fixed. So it is more cost-effective and makes more sense to test the product manually.

Nowadays, most software projects are so complex that it is not feasible to plan, design, and think over all the technical details in advance. The projects are usually long-term, while technologies and business needs change fast. It has become extremely important for software products to respond to customers' feedback fast. So instead of having big rare releases, software companies have to have rapid delivery cycles and release new versions of software as quickly as possible. Today, some

companies deliver new features and bug fixes many times a day and even a few times a minute.

Testing each version manually can take a lot of time, and that's the main reason why test automation has become so important. There are other reasons as well which will be considered further in this paper.

1.2 The Cost of Software Complexity

With every new version of the software, new features are added. As new features are added, the software becomes more complex, and when the software becomes more complex, it becomes harder and harder to add new features to it without breaking anything. This is especially true when there is pressure to deliver the new versions rapidly and not investing enough time to plan and to improve the quality of the code. Some of this added complexity is unavoidable. It would have existed even if we carefully planned and designed the entire software ahead. This is called inherent complexity. But most of the time, most of the complexity in software exists because features were added quickly without proper design; lack of communication inside the team; or due to a lack of knowledge, either about the underlying technology or about the business needs. Theoretically, this is about the business needs. Theoretically, this complexity could be reduced if the software was carefully planned in advance as a whole, but in reality, it is a natural part of every software project. This type of complexity is often called accidental complexity. Any complexity, be it inherent or accidental, comes with a cost. This cost is of course part of the overall cost of developing software, which is mainly affected by the number of developers and testers, multiplied by the time it takes for them to deliver the software. Accordingly, when the complexity of a piece of software grows, its cost increases because it takes more time to test everything, and also it takes more time to fix (and retest) the found bugs. Accidental complexity in particular also makes the

software more fragile and harder to maintain and therefore requires even more time to test and more time to fix bugs [2].

1.3 Refactoring

Refactoring is a must-have practice in software development when following Agile methodology. It is a process of improving the design and code of the software without changing its functionality. Refactoring also allows keeping accidental complexity under control.

Though refactoring is an important software development practice, it still can break the software functionality and bring defects. That is why extensive testing is required after each round of refactoring. If you have only manual testing on the project, it means that you will need to repeat the same testing routine quite often, which will cost time and money on one hand and will affect test team motivation on the other hand. And in this case test automation becomes really helpful.

1.4 Test Automation and Process Improvement

Test automation has its relationships with all other aspects in the software development cycle. Besides quality and productivity, test automation is also related to the architecture of the product, the business processes, the organizational structure, and even the culture (see Figure 1.1).

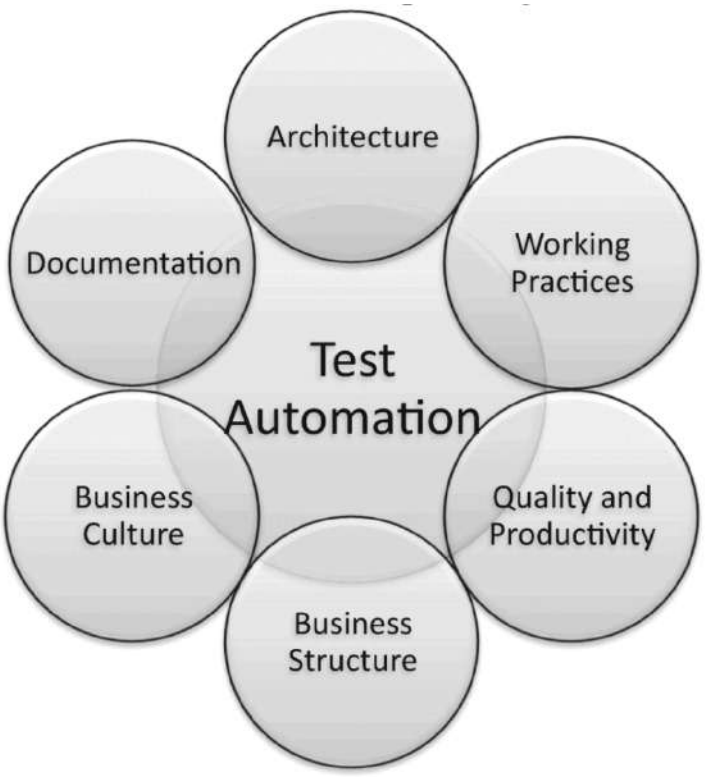

Figure 1.1 - Test automation dependency on software development aspects [2].

All mentioned aspects can affect the test automation practice within the company. But the dependency is mutual - they can also bring a positive influence on test automation processes.

1.5 Test Automation Tools

Let’s have an overview of the most popular test automation tools.

1.5.1 Selenium WebDriver

Selenium WebDriver is the most popular test automation tool. It is mainly used for automating tests for web applications on the User Interface (UI) level. The tool allows to imitate mouse and keyboard actions on behalf of a user and also to retrieve data that is displayed on the screen.

The main advantages of Selenium WebDriver are the following:

- it’s free
- supports multiple browsers
- allows using different programming languages (Java, Python, C#, JavaScript)

The advantages listed above actually make Selenium WebDriver the most widely used test automation tool. However, it also has some disadvantages, which are:

- limited support for other interfaces except for browser interface
- is designed mainly to be used from code

Let’s consider the specifics of how Selenium WebDriver works, see Figure 1.2. It is composed of two interchangeable parts - language binding and browser driver. The language binding is the code library. It provides classes and methods that you can use in the tests. There is a different language binding for each supported programming language (Java, Python, C#, JavaScript). It communicates with the browser driver using a dedicated JSON wire protocol. After getting a request from the language binding, the browser driver invokes a relevant operation in the browser. There is an own driver in each browser. You can run the same test in different browsers using the corresponding browser driver.

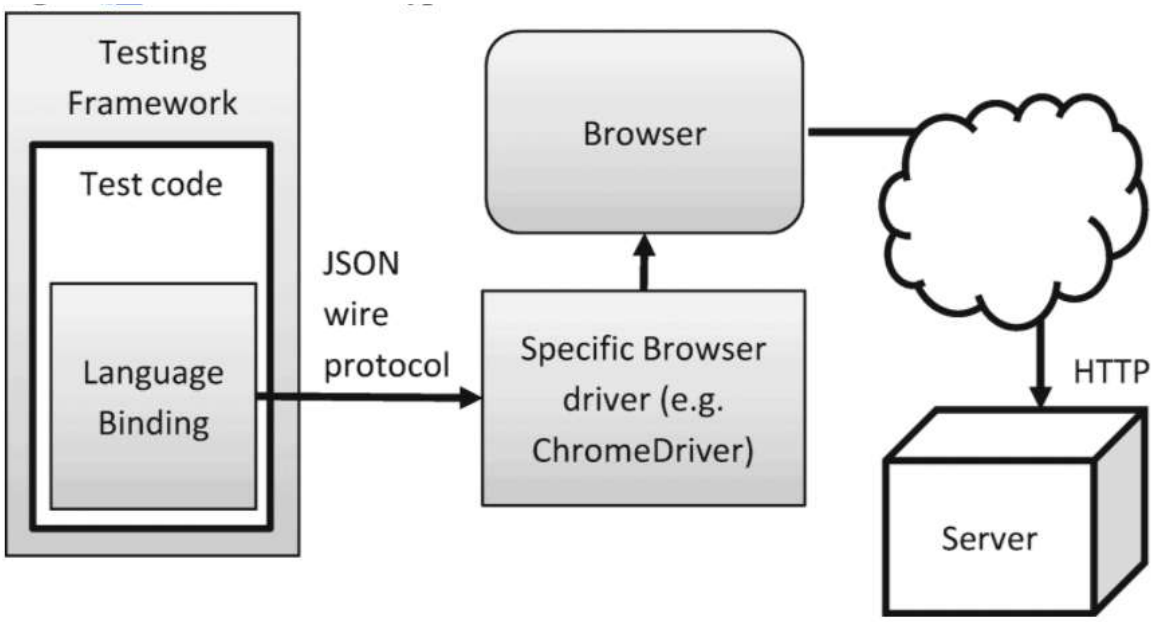

Figure 1.2 - Typical architecture of a Selenium WebDriver test.

Selenium WebDriver is also commonly used from unit testing frameworks. It has a quite flexible architecture, so various special tools can be integrated with it, such as Selenium Grid (tool for on-site cross-browser testing), BrowserStack (cloud-based testing provider), SauceLabs (cloud-based testing provider), etc.

1.5.2 Cypress

Cypress.io is an open-source test automation tool, created in JavaScript language. Cypress could be used for different testing tasks like end-to-end testing, integration and unit testing, tests execution in CI and debug. It provides extended capabilities to operate with browsers and act like a human in GUI and as a server on the back-end.

In addition to the test runner, Cypress provides a GUI which allows users to run tests, see real-time test execution, catch screenshots and videos, debug, logging and more, Figure 1.3. It allows you to create automated scripts for very UI interactive applications and deal with actions like manipulating the DOM, do assertion for elements visibility, input data into fields, directly redirect to a different page and event work as a mocking solution.

Cypress is based on very popular JavaScript tools like Mocha and Chai and provides additional features on its functionality. This provides capabilities to build and deliver a modern Web application with build-in quality and developers confidence.

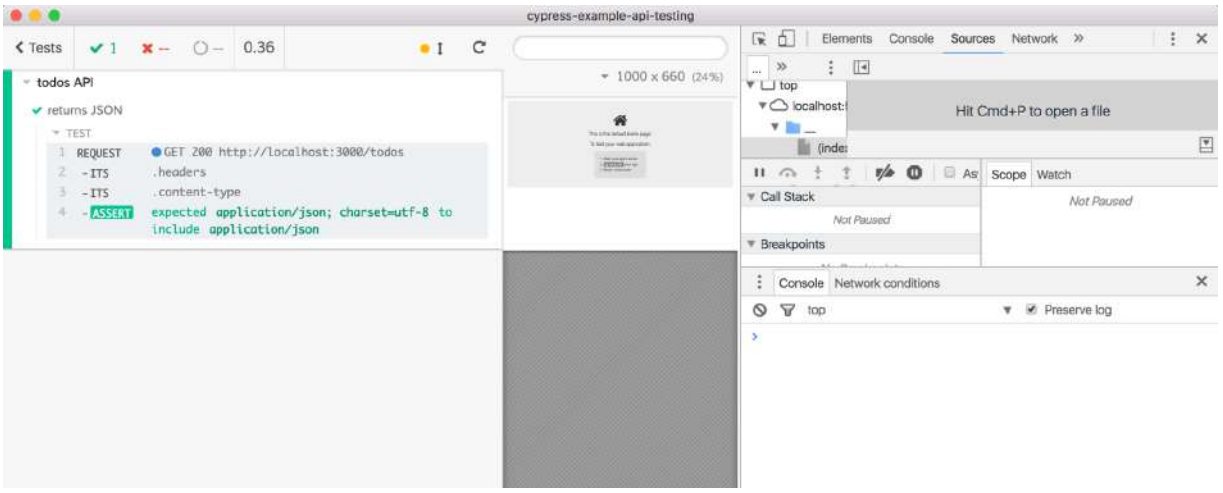

Figure 1.3 - Cypress tool GUI [31].

1.5.3 Browser's built-in Developer Tools

Many tools for test automation on the UI level (especially commercial ones) come with a special tool for identifying elements of the interface and their properties. Selenium WebDriver does not have it. However, all modern browsers have a built-in toolset called Developer Tools (usually opened by pressing F12). The Developer Tools include the DOM4 explorer, which allows you to identify the elements and their properties. In Figure 1.4, you can see an example of the DOM explorer in the Chrome browser.

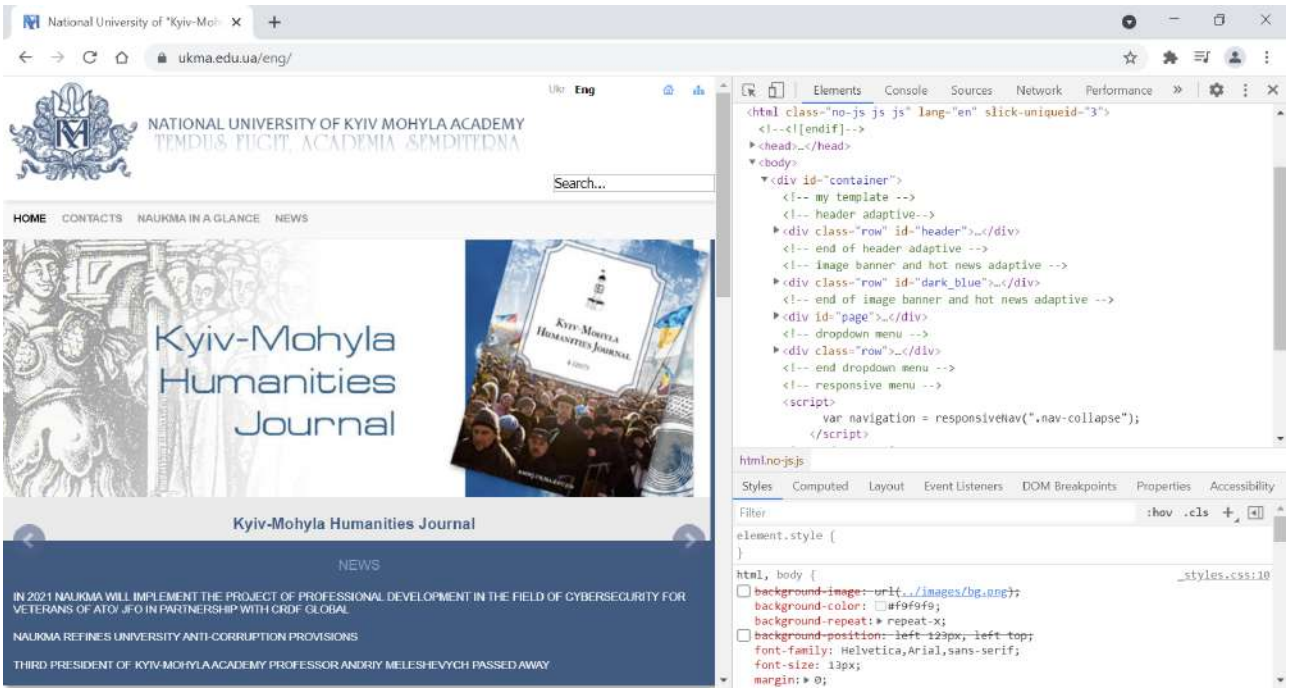

Figure 1.4 - DOM explorer in the Chrome browser

So browsers' built-in Developer Tools are commonly used by test automation engineers to locate UI elements properties for accessing them from tests.

1.5.4 Visual Testing Tools

Visual testing is a way to test the software by comparing the actual displayed image against the predefined template. UI automation tools such as Selenium WebDriver test the application on the UI level, but they do not verify the appearance of the UI elements (colors, sizes, shapes, etc.). Nowadays the look and appearance of the UI elements are important because they affect usability and user experience.

UI automation tools rely not on elements' exact location, but on their size, colors, etc. Obviously, these things can change frequently and thus fail the test. But sometimes verifying the appearance of the UI can be very valuable, in addition to regular functional tests.

One of the best tools in this area is Applitools Eyes. The Sikuli open source project is also popular, but it uses visual object recognition more for identifying elements rather than for visual testing. Another known tool for visual testing is Chromatic, it is specifically tailored for the React framework.

We must understand that strict pixel-by-pixel image comparison is not reliable and stable enough (due to things like screen resolution, etc), so visual testing tools usually use quite a smart image comparison technique to overcome limitations. Visual testing provides an API that you can integrate with your functional tests to take screenshots in relevant moments in the test, either of the entire page or of a specific element. During the first test run, the images are saved, and they will be used as reference images during the next runs. So the next time a test is run, it will be comparing new screenshots against reference images. The differences will be highlighted, and then you can either exclude the highlighted segment from the next comparison or update the reference image. Sometimes, maintaining visual tests may take lots of effort, especially if UI changes frequently or if you have a lot of images in your tests. In this case, visual testing is not a proper choice of test automation tool.

Despite the issue described above, visual testing tools can really help with cross-browser and cross-platform testing. Nowadays there are tens and hundreds of browsers, operating systems, mobile devices, and you may need to test your software on many of them. If you have short release cycles, it will not be feasible to test on all those platforms manually. Applications are designed to be cross-browser and cross-platform and should look the same on platforms and devices, but sometimes they don’t. Different browsers and operating systems have different rendering engines and that can cause differences in UI appearance. Applitools Eyes or similar tools allow to control the level of accepted variations, and that can overcome the legitimate differences between browsers from one hand, but still catch more severe rendering problems on the other hand.

1.6 Test Automation and Architecture

Any test automation solution should be considered a software product. It means that it also must have its own architecture. Under architecture, we usually mean the bunch of high-level decisions that influence the entire solution, and the more you work on the solution the harder it is to change them.

The architecture of a test automation solution defines how tests are coded, how they interact with the system under test (SUT), how they are run, etc. And it is really important to take into account the architecture of SUT itself. Automated tests must also be reliable and isolated, and that's another important task for their architecture.

In general, when working on test automation architecture decisions, you must consider the following questions:

- who will develop automated tests?
- who will maintain the test automation solution and how easy will it be?
- who and when will be running the test?
- which components of SUT will be covered by tests?
- how long can tests be executed?
- how much effort will be needed to develop new tests?
- how easy will it be to understand test results and investigate failures?

Usually, when we start working on test automation architecture, we define the areas of SUT that we want to cover with test automation. This is important to understand that it's not always feasible to test the entire system end to end and keep tests reliable. Therefore knowing the system under test, its architecture, and understanding the input/output combinations are essential when outlining architecture for a test automation solution.

Now let's consider what is an automated test. We can define an automated test as a computer program that sends inputs to another computer system (the system under test); compares the output sequence, or part of it, to some predefined expected result; and outputs the result of that comparison [2]. Figure 1.5. shows that description of an automated test.

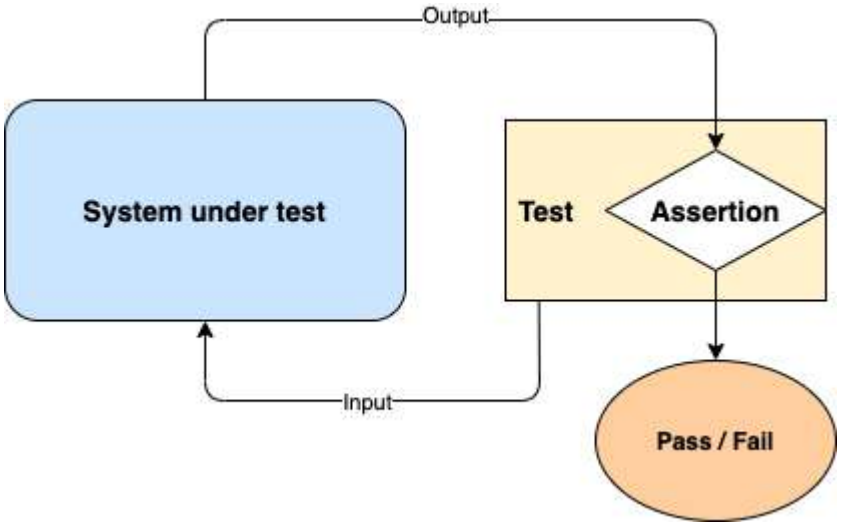

Figure 1.5. Description of an automated test.

One of the main test automation challenges is the fact that very few systems today are really "stand-alone". Most of the systems have multiple dependencies on external services or other systems which are not under our control. That is why it is difficult to draw a clear line between where our system ends and another one starts. So we are concerned: what components or services we must test? It is not a big concern for manual testing, because manual testers mainly act as the end-users, and test that what they see makes sense. If they encounter problems with external services, they usually understand on which side the issue is. Also, manual testers take into account that often the outputs of SUT depend on inputs that it receives from external services, and testers can define if they are correct or not. But it works differently for test automation. For automated tests, we must be able to define a deterministic expected result. This means that we must control all the inputs of the system that may affect the outputs that we want to verify, including inputs from those

external services. The case is complex because in modern software systems there is not just one input and one output. There are sequences of inputs and streams of outputs and many independent sources. Moreover, the inputs and outputs are often so tightly related, so it is hard to distinguish them from each other. This makes planning test automation architecture to be a really hard task. And you must find a solution to draw a clear line between SUT and external services and handle multiple dependencies.

If we would have control over all input sources, then we could develop automated tests with deterministic expected results. Controlling inputs received from external systems can become feasible if you, for example, simulate them. Tests must control everything that we consider as inputs to the SUT and that may affect the outputs that we need to verify.

1.7 Continuous Integration and Continuous Deployment

Nowadays every software project assumes having Continuous Integration (CI) - a must-have development practice when developers integrate code into a shared repository as frequently as possible. It’s a standard for the industry. And recently it has been expanded with Continuous Delivery and Continuous Deployment (CD) which have become essential for mature software projects in the Agile environment.

Just about 10 years ago, delivering a new version of the software required huge efforts (for example, manufacturing of a physical CD ROM disc). Today we download software from the Internet or get updates for web applications automatically. This means that users can get new versions quite rapidly (and automated tests can even speed up this process). Most of the software products are updated via a centralized deployment and distribution system. However, the deployment process itself needs to be automated. If the deployment process has manual steps, it takes more time and increases the risk of mistakes.

Even if you run your test automation in an isolated environment, you still should do CI for it. Then it also forces you to automate the deployment process, because it will help to automate the process of deploying new versions to production more easily. Automation of the entire deployment process is called Continuous Deployment (CD).

Modern tools and approaches allow automating the entire deployment process. But in real life software producers prefer to decide what goes into Production and when. It's not a technical question, but a business decision. The good practice is to deploy a new version to a production-like environment first and perform some manual testing there. So, in this case, the entire process is automated, but the final step requires manual intervention. This is called Continuous Delivery.

Often companies that produce software for non-critical domains use the Continuous Deployment practice. Sometimes they have small releases for small groups of users, and in case of failures, they fix them fast. As for software applications from critical domains (such as medicine, aviation, etc.), every fix can take a long time because it requires thorough verifications. So such companies usually choose the Continuous Delivery approach.

All the approaches and practices described above include the test automation component as a must-have part.

1.8 Issues with traditional automation testing tools

In the last decade, the speed of testing and release to production dramatically changed from months and years to weeks and hours. Nevertheless, automated testing as an obligatory attribute on any CI/CD system remains complicated and challenging.

Visual testing remains complicated for automation and often additionally covered by manual exploratory testing.

Traditional methods of automated visual testing by verification of CSS styles and specific HTML page elements during functional testing could work for a limited number of components but cannot be used for verification of all visual elements. Also, this method is time-consuming to write and support.

For example, each component could be verified to be visible, right coordinates, height, width, color, and others. This means, one component could have 5 visual assertions to cover. So, a test for just 10 components will lead to 50 lines of code.

As an alternative to traditional methods, a snapshot testing tool is used to capture the page image and compare it with the reference image. Such a method has the benefits of speed recognition and could work if the test page is on the same resolution. On the other hand, this method produces false-positive results if the page is just slightly updated, see Figure 1.6.

Figure 1.6 - Example of false-positive results with snapshot comparison [5].

In order to respond to the market demands to produce and release software with even more speed and with build-in quality, automated testing tools need to adjust and utilize new emerging technologies like artificial intelligence and machine learning.

This work investigated an option to extend already well-defined function testing with machine learning for efficient and robust visual testing.

2. Design of an automated testing system using artificial intelligence tools

2.1 Architecture documentation

The C4 + 1 model for software architecture documentation has applied in this paper.

The C4 model approaches documenting and designing software architecture by diagrams. The model has a granularity of four diagrams that represent the whole system. Additionally, a deployment diagram has been added to represent the infrastructure qualities of the solution [29].

The C4 Model represents four levels of diagrams:

- System Context diagram (Level 1): shows the highest level of the system and how it corresponds to users and other systems. It is proposed as a quick representation of the system and suitable for presentation for different stakeholders. It represents the system as the big picture diagram [30].

- Container diagram (Level 2): separates the system into interconnected containers. Containers are deployable and executable subsystems. It is designed to provide people who need additional technical details on how the system is set up [30].
- Component diagram (Level 3): splits containers into interconnected components, connects them to other containers or other systems. It is intended for developers who are not yet familiar with the system and going to develop on it [30].
- Code diagram (Level 4): provides more detailed information on how the implementation looks in the actual code. It is designed to clearly show some specific details about the most complex parts of the system [30].

The C4 model's four levels are sufficient to describe complex Software Architectures. In practice, the first three levels are only most applicable, while the fourth level, like the actual code, is done by engineers who need to review and understand.

2.2 System architecture diagrams

2.2.1 System context diagram

The designed Automated testing tool has three actors who contribute and get results from the tool. System Context diagram represented in Figure 2.1.

Developer and/or QA engineers (actor) - create test cases for Web Systems with user interface under test. After that, they create automated tests with the usage of Automation testing tools. Automated tests have assertion and expectation for each case so automated cases have exact criteria to pass or fail. Automated tests merged to the tool repository.

UX Designer (actor) - creates markup for Web system and uploads images per component into Automation testing tool. Uploaded images later used as verification source during Web system testing.

Product Owner or other stakeholders (actor) - optionally could be notified with a report about automated test execution results. Report generation could be limited to specific environments such as Staging or Pre-production to avoid unnecessary false-positive results during development.

Automated testing tool - executes automated cases and runs tests on a system to test across different environments. The execution could be started manually or scheduled automatically on the event (successful deployment) or a specific time.

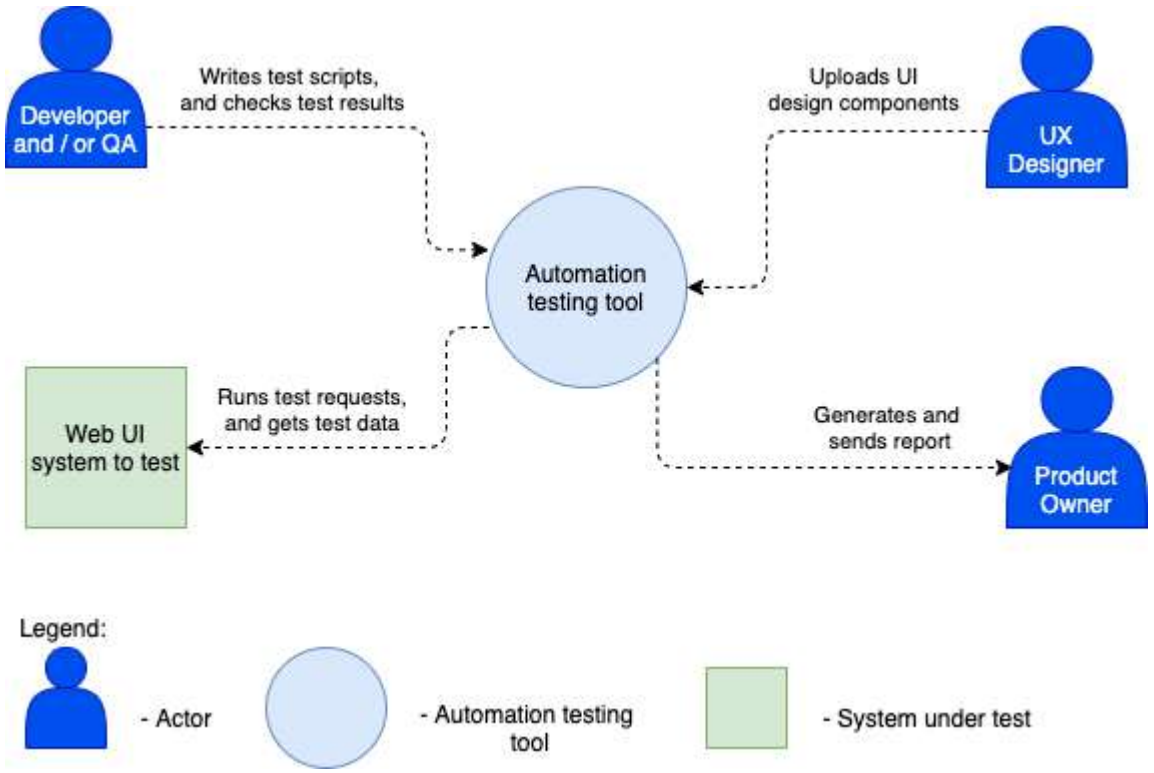

Figure 2.1 - System context diagram.

2.2.2 Container diagram

The container diagram shows the Automated testing tool in the container view that makes the system as presented in Figure 2.2.

Test framework - contains automated tests implemented by developers or QA engineers, test runners, test utilities. It executes created tests against target Web applications, does assertions, and generates reports. In this paper cypress.io, an end-to-end framework was selected.

For visual testing validation, the framework is extended by custom functions to take DOM snapshot and send it to the Components screenshot engine.

Components screenshot engine - this container renders DOM snapshot and splits it on components images for further comparison with baseline image component by ML engine. Implementation of this component outside of this work.

ML engine - meant for comparison of rendered components images with UI components design. This container includes a convolutional neural network (CNN) with a pre-trained model from UI components design storage. The container compares images and returns results to the test framework.

UI components design storage Web app - this application is for UX Designer to upload and store UI components design and mark them by proper labels. This work used simplified storage with folders structure of files. Further, a more comprehensive solution could be added to the system.

UI components design storage - storage for design components. This work used storage with folders structure of files with folder name as a label. Later this storage could be updated to a more comprehensive solution with a database.

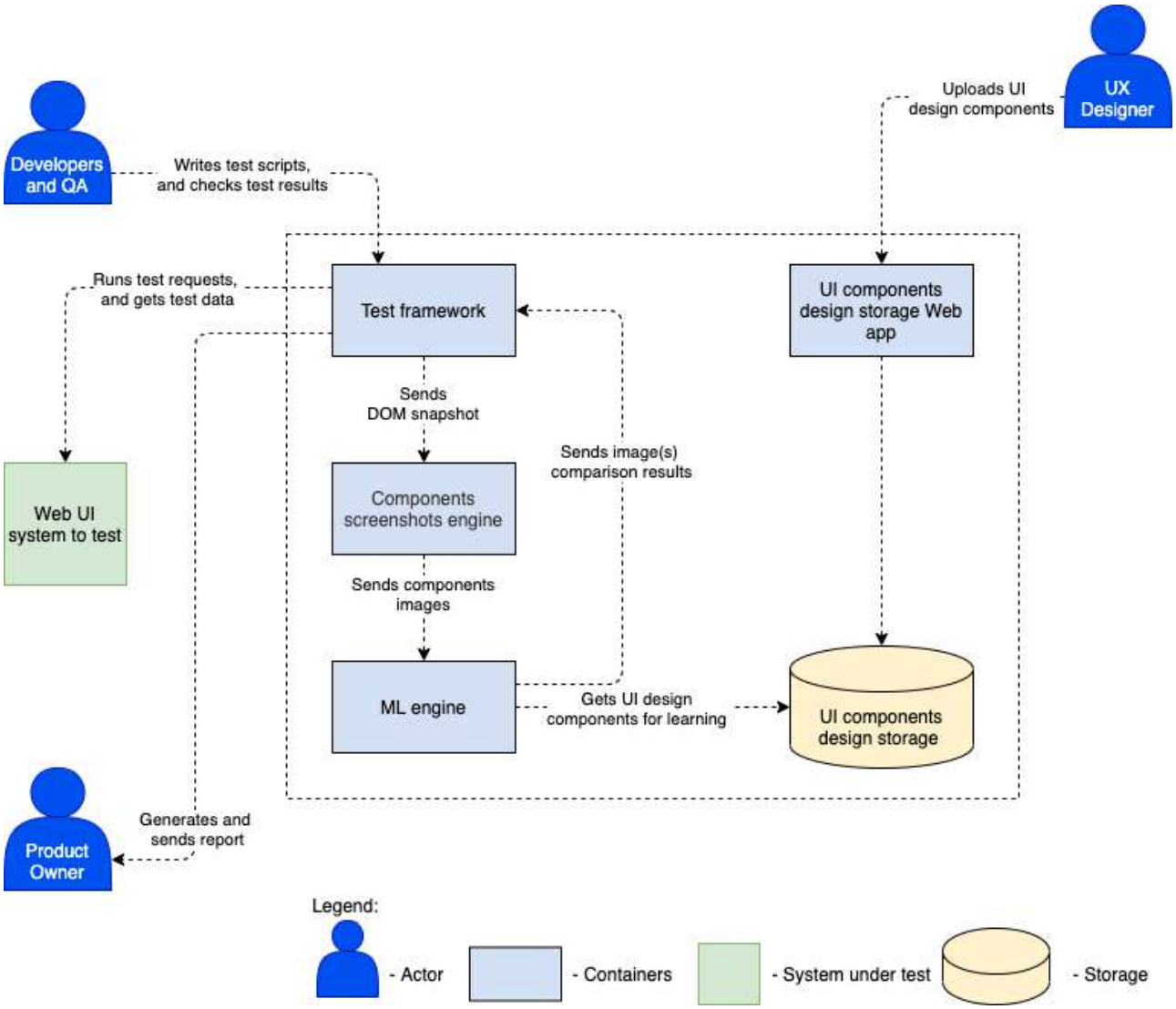

Figure 2.2 - Container diagram.

2.2.3 Component diagram - Test framework

As a core of the Test framework, the Cypress.io end-to-end testing framework has been used. Cypress.io provides comprehensive functionality for functional end-to-end testing in web web-browsers, test runner, assertions, and reporting.

Cypress has communication with a browser on two protocols. Websocket used for test execution. HTTP used via proxy for network monitoring and mocking or stabbing.

For the purpose of this work, the test framework needs to be extended with custom plugins to extend available functional capabilities with additional visual

testing and machine learning validation. Another testing framework could be used as well depending on the testing targets and scope.

With extended custom steps, the Cypress framework takes a DOM snapshot of the page and sends it to the Components screenshot engine for further rendering and validation.

ML engine sends back image comparison results to the test framework to pass or fail depending on the ML engine comparison result.

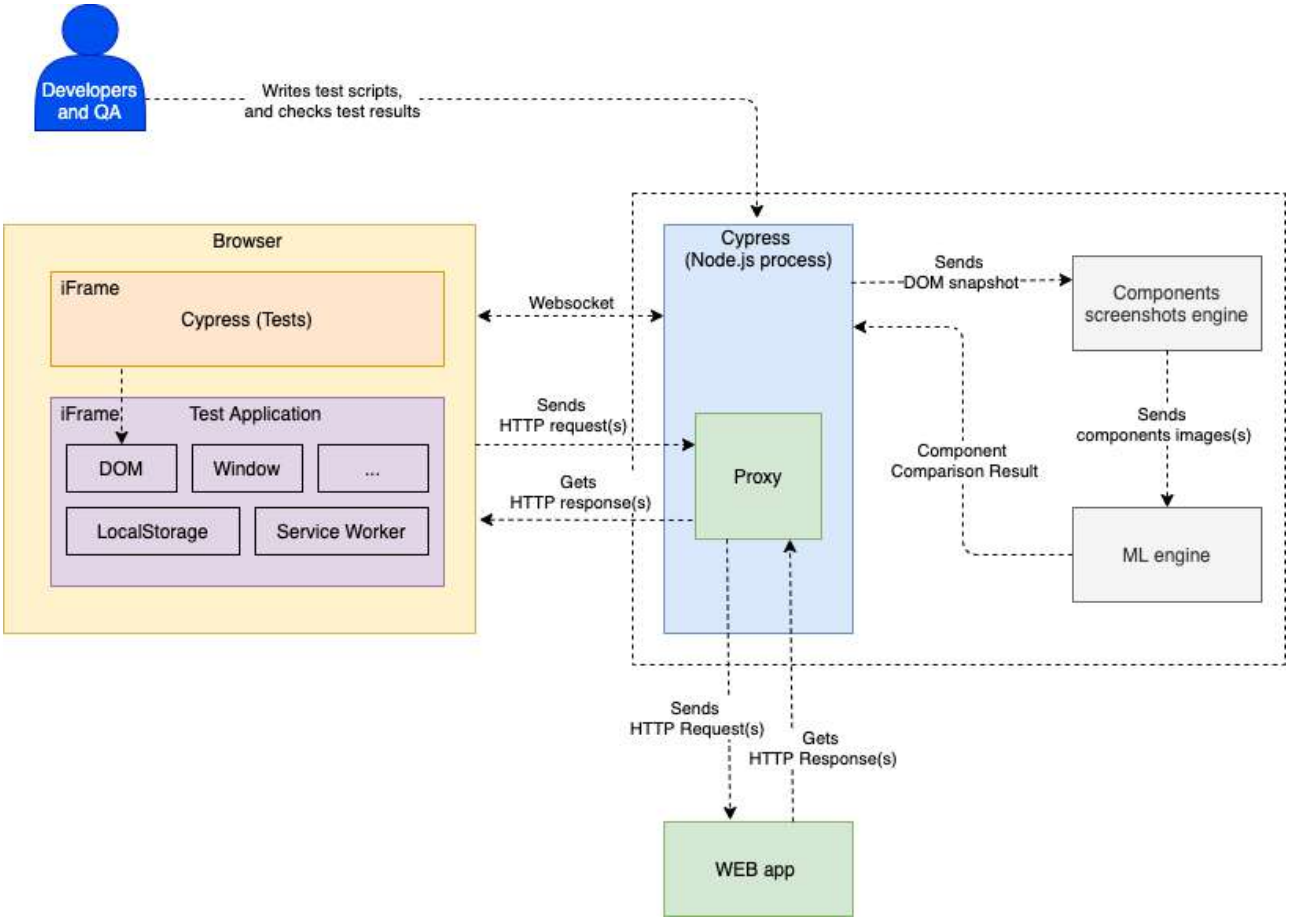

Figure 2.3 - Component diagram - Test framework.

2.2.4 Component diagram - Components screenshots engine

Components screenshots engine meant for render of the DOM snapshot and creation of components images for further validation. Snapshot render could be parallelized and executed for different browsers, resolutions, and mobile device emulation.

Components image processor sends to ML engine for validation and comparison with UI design components.

Implementation for Components screenshots engine outside of this work.

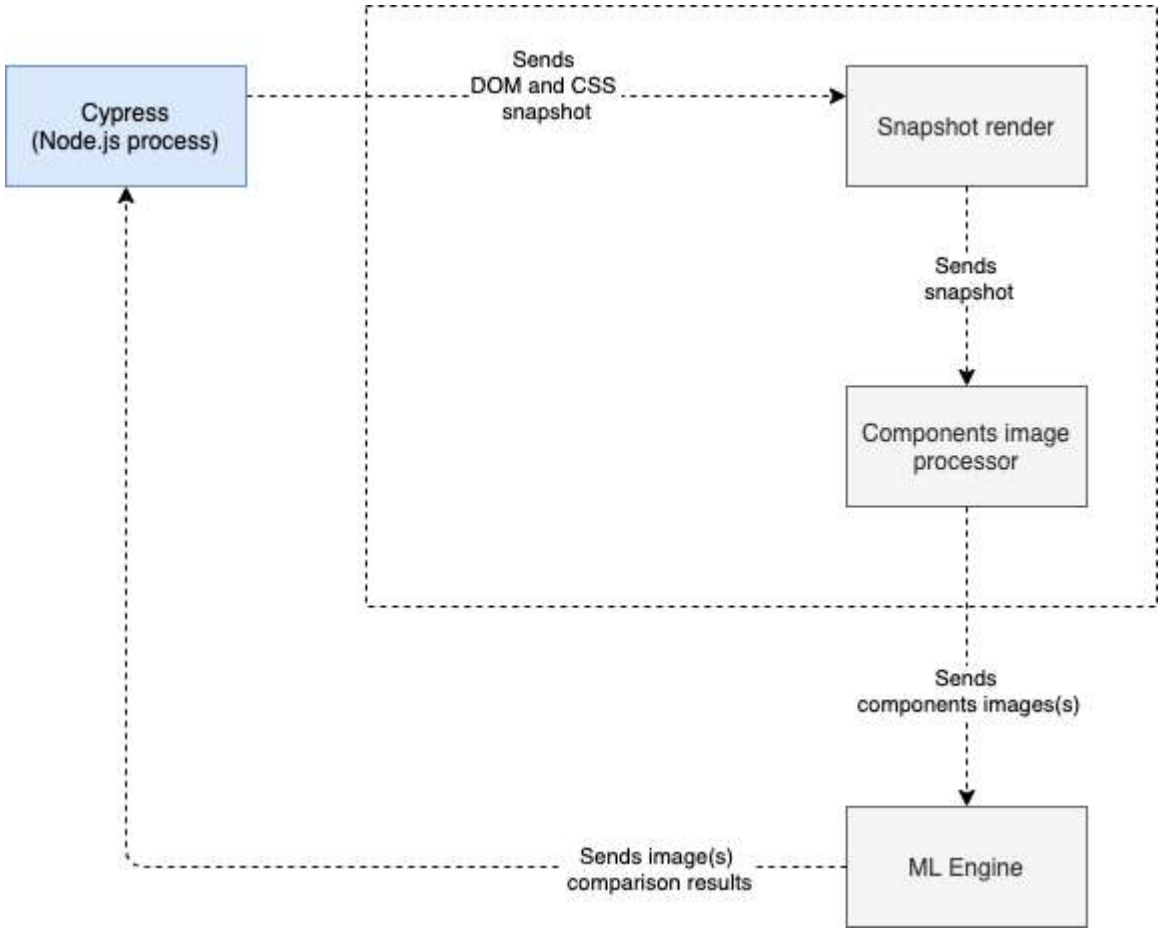

Figure 2.4 - Component diagram - Test framework.

2.2.5 Deployment diagram

The designed system should meet the following architecture significant requirements (ASRs):

- Performance ;
- Scalability;
- Stability;
- Affordability
- Maintainability;

Mentioned ASRs attributes could be achieved by cloud infrastructure. Amazon Web Services deployment diagram depicted in Figure 2.5.

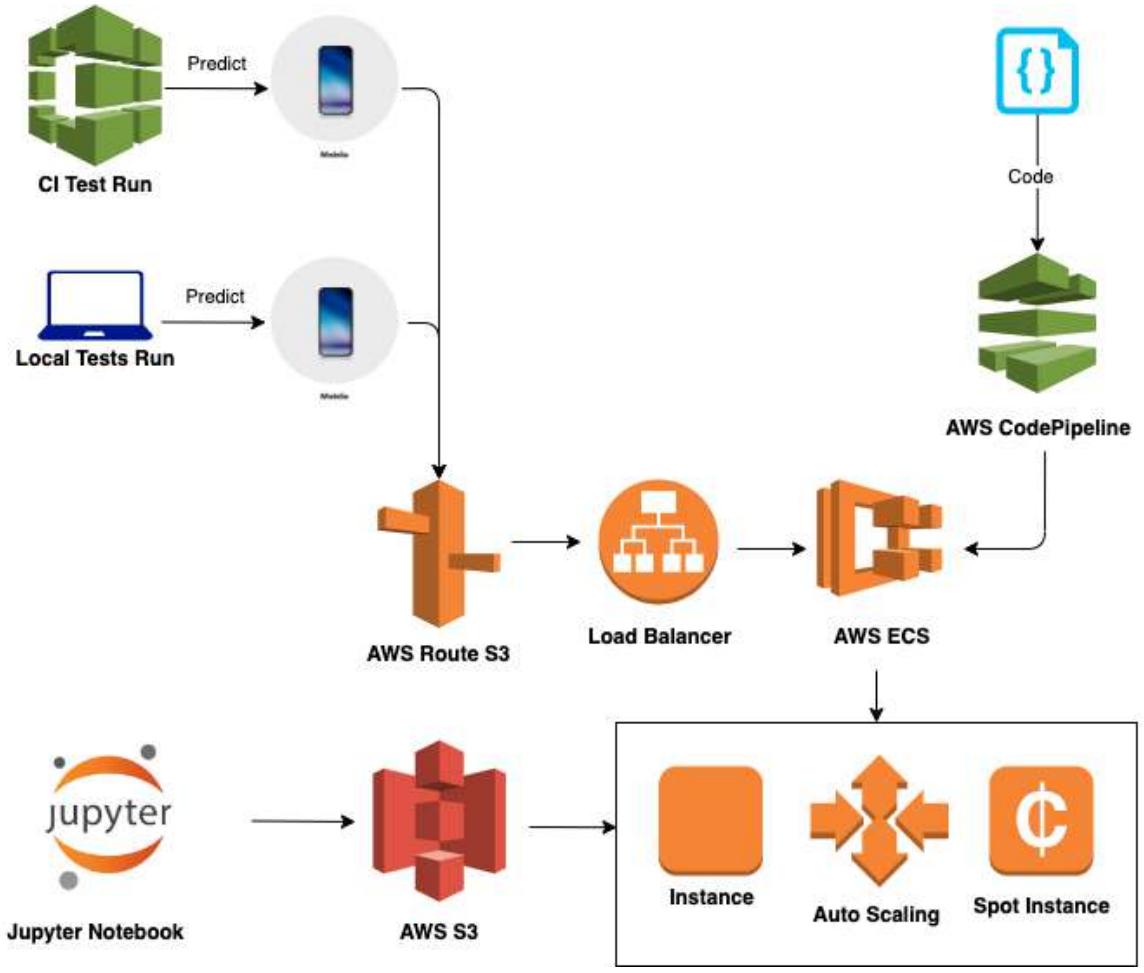

Figure 2.5. Deployment diagram.

3. Implementation of an automated testing system using artificial intelligence tools

3.1 Testing website

For the purpose of the web application under test, a test site was published based on the Wix site constructor. The Wix constructor provides customizable templates and a drag-and-drop site builder with features like application, plots, galleries, different fonts, and other features. Such sites have a rich and complicated GUI and could be selected for automated testing purposes.

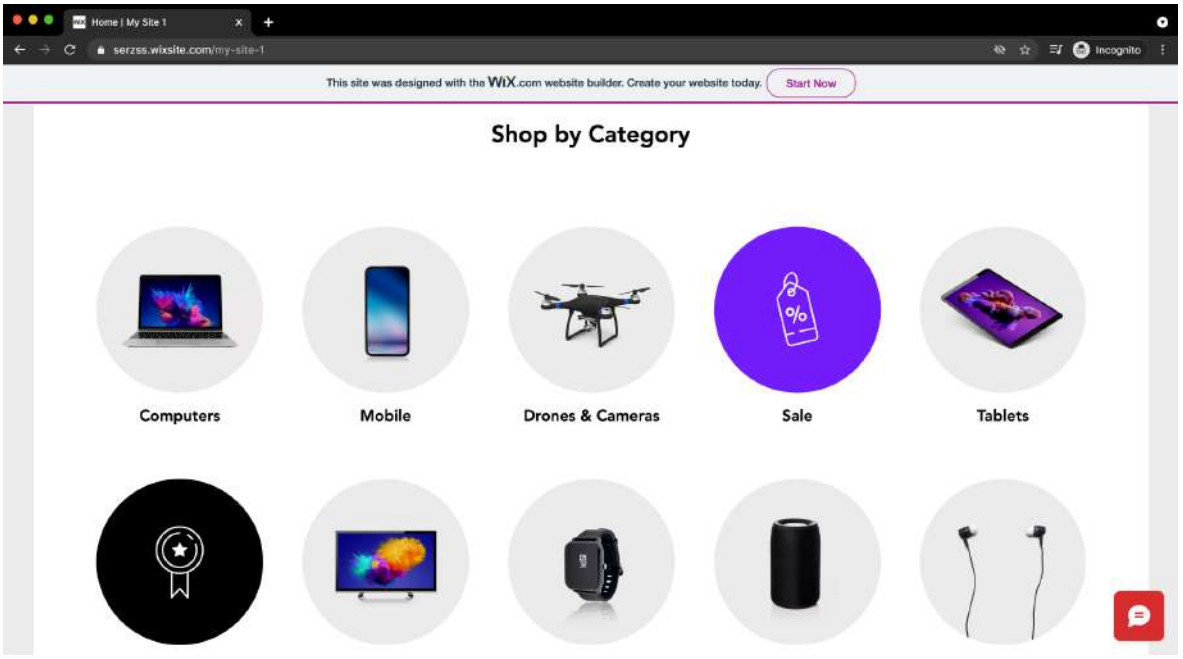

Figure 3.1 - Website as the system under test.

3.2 Functional test extension for components comparison

In this paper, a Cypress open-source test automation tool was used. This provides capabilities to extend base functionality by providing a platform for

custom-created plaguing and functions. With the use of a custom plugin, the functionality of the tool was extended to include a component comparison verification into a functional test.

Common Cypress test for the visual component test will use CSS properties and ‘have.css’ assertions. This approach is very complicated for implementation due to the significant number of CSS properties and its dynamic nature. Maintaining such cases is another challenge and quite a routine and time-consuming task. Example of such a test depicted in Figure 3.2.

```
describe( title: 'Functional test ', fn: () => {
  it( title: 'should check Best Sellers component', fn: () => {
    cy.visit('https://serzss.wixsite.com/my-site-1');
    cy.get('#best-sellers').should( chainer: 'have.css', value: 'have.width', match: '160px' );
    cy.get('#best-sellers').should( chainer: 'have.css', value: 'have.height', match: '160px' );
    cy.get('#best-sellers').should( chainer: 'have.css', value: 'background-position', match: '50%');
    // and more assertions
  })
})
```

Figure 3.2 - Functional test for visual components.

This work discovered another approach to visual testing with the usage of CNNs for component image comparison. The custom step takes a component snapshot (or the entire page) and compares a snapshot by CNN trained model based on UX designs. If the component comparison passes, by the model, it will mean that the web application looks the same for users and meets requirements set feature implementation and design. If differences are found, then the case should be investigated for visual differences like layout, fonts, color, or other visual attributes. Example test depicted in Figure 3.3.

```
describe( title: 'Component comparison ', fn: () => {
  it( title: 'should match Best Sellers component', fn: () => {
    cy.visit('https://serzss.wixsite.com/my-site-1');
    cy.get('#best-sellers').compareComponentSnapshot();
  })
})
```

Figure 3.3 - Test with visual component verification.

3.3 Implementation of Convolutional neural network (CNN) for components comparison

3.3.1 CNN's overview

Convolutional neural networks (CNNs) have been developed on the basis of the brain's visual cortex research, such networks highly used for image recognition starting in the 1980s. In the last decade with the increase in computational power and huge amounts of training data, CNNs have become able to achieve really great performance on some very complicated visual tasks. Such networks are used in image search services, self-driving cars, and automatic movie classification systems.

One of the most important building blocks of CNN is the convolutional layer. Neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields [32]. Consequently, each neuron in the second layer is connected only to neurons located within a specified rectangle in the first layer, Figure 3.4.

CNN architecture focuses the network on little low-level features in the first hidden layer, then mounts them into bigger higher-level features in the next hidden layer, and so on each following layer. Such structure is typical in real images because CNN's suit very well for image recognition.

Taking CNNs advantages in image processing, this model was chosen as the basis for components comparison solution.

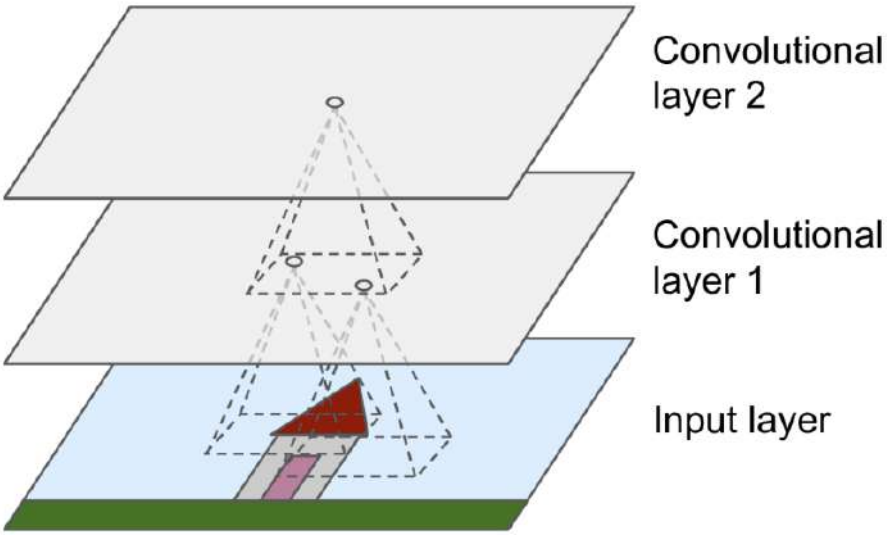

Figure 3.4 - Convolutional layers.

The neural network for current work has the following requirements:

- The machine learning approach is used as it perfectly suits work with unstructured data like images.
- The model receives an image at the input and returns a subject.
- The model will use the project training collections to learn.

The overview of the selected CNN design for this work is depicted in Figure 3.5.

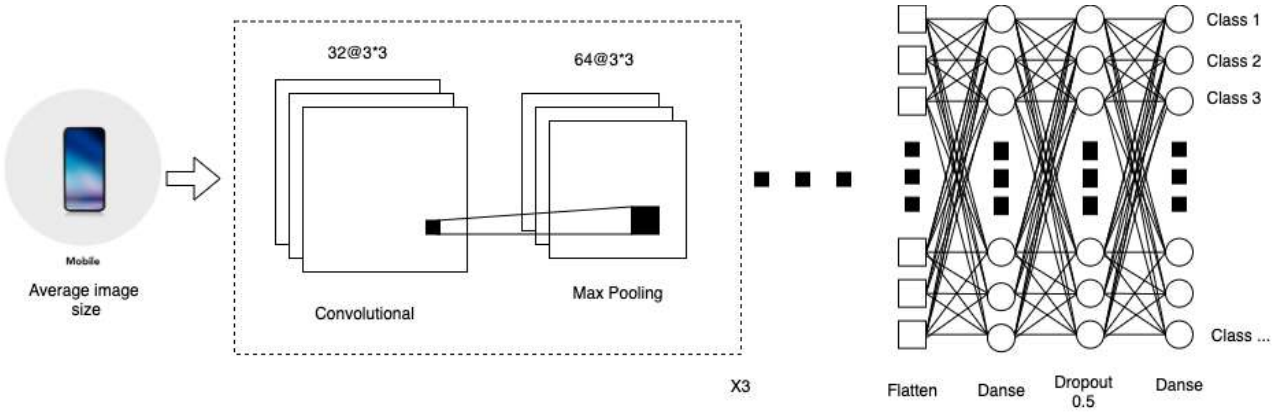

Figure 3.5 - Convolutional neural network overview.

3.3.2 Data set review

In this work, the TensorFlow open-source library and Python as script language have been used.

TensorFlow library has been designed to train and run deep neural networks for different purposes and scales like image recognition, written digit classification, natural language processing, and others. The trained model supports production prediction at scale.

TensorFlow uses the Python language for high-level programming abstractions. Nodes and tensors are Python objects, as TensorFlow applications are. The actual math operations are not performed in Python, C++ used for this, Python just connects traffic between components.

TensorFlow applications could be run on almost any platform: local machine, cloud, mobile device, CPU, or GPU. The trained model could be deployed on almost any device to serve for predictions.

A data set for training has train and test folders are grouped by categories for further feeding into the model. The categories used in the work are depicted in Figure 3.6.

```
train_path = data_directory+'/train'
train_augmented_path = data_directory+'/train_augmented'
test_path = data_directory+'/test'
```

```
([dir for dir in os.listdir(train_path) if dir != ".DS_Store"])
```

```
['sale',
 'computers',
 'shop_now',
 'curbs-side_pickup',
 'low_prices',
 'drones',
 'availability',
 'werable',
 'free_shipping',
 'best_sellers']
```

Figure 3.6 - CNN train categories.

```
free_shipping_file = train_path+'/free_shipping'+'/free_shipping1.png'
train_image_1 = imread(free_shipping_file)
plt.imshow(train_image_1)
```

```
<matplotlib.image.AxesImage at 0x7fb8a3cf2c10>
```

Figure 3.7 - Original component design image for trains.

According to the system design components, images could be produced by a designer in different shapes, resolutions, and sizes, Figures 3.6 - 3.7. As a result, the data set has differences in image shape and size that are not suitable for model trains and should be converted to one value.

For this purpose, additional steps were added for image processing. The train folder scanned for image shape and defined an average image size for all further actions, code for this is presented in Figures 3.8 - 3.9.

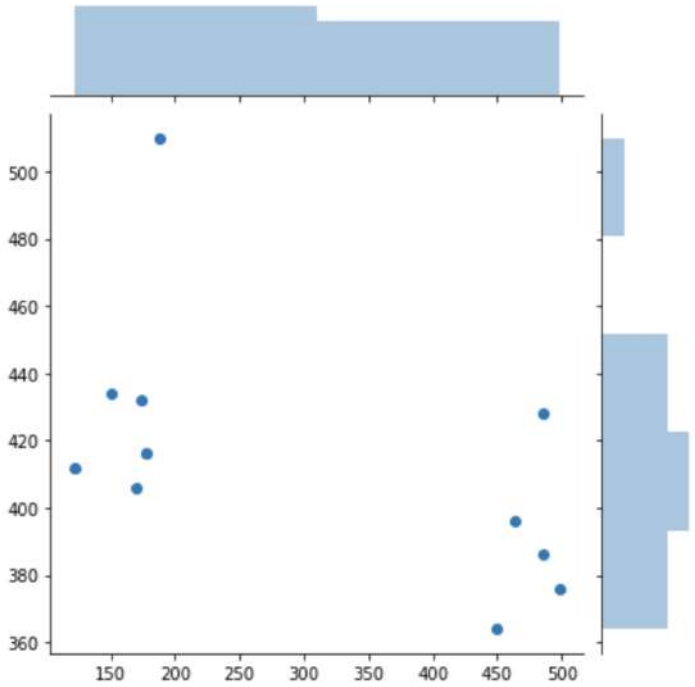

Figure 3.8 - Image sizes in the dataset.

```
dimention1 = []
dimention2 = []
for subdir, dirs, files in ([dir for dir in os.walk(train_path) if dir != ".DS_Store"]):
    for filename in files:
        file_path = subdir + os.sep + filename

        if file_path.endswith(".jpg") or file_path.endswith(".png"):

            image = imread(file_path)
            d1,d2,colors = image.shape
            dimention1.append(d1)
            dimention2.append(d2)
```

Figure 3.9 - Define average image size.

```
height = np.mean(dimention1)
width = np.mean(dimention2)
image_shape = (height,width,3)
```

```
image_shape
```

```
(306.0, 414.54545454545456, 3)
```

Figure 3.10 – Set average image size.

3.3.3 Artificially expanded data set

During Web application development UX designers would upload components design images to the system. Each component will have one or several images. Such a number of images could be sufficient if the image for comparison will have very similar quality. In our case, the system should have the possibility to compare components rendered on different browsers, resolutions, and devices.

To have a robust model to compare images from different resources the data set should be significantly extended. For this purpose TensorFlow built-in functions to automatically generate artificially extended sets of data have been used.

A new extended data set has been created by additional image manipulation which includes rotation, resizing, and zoom. The ImageDataGenerator used to create augmented images in an automated way, figure 3.7.

Original image manipulated with parameters:

- rotate the image on 5 degrees;
- shift the image width by 5%;
- shift the image height by 5%;
- image normalizing;
- cutting away some part of the image by 5%;
- zoom the image on 5%;

As a result, an augmented data set consists of a significant number of manipulated images that could represent real images for comparison. The example presented in Figure 3.8. Such a data set perfectly fits for model train porpoise.

Code for augmented image generation for each category presented in Figure 3.11.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
image_gen = ImageDataGenerator(rotation_range=5,
                               width_shift_range=0.05,
                               height_shift_range=0.05,
                               rescale=1/255,
                               shear_range=0.05,
                               zoom_range=0.05,
                               horizontal_flip=False,
                               fill_mode='nearest',
                              )
```

Figure 3.11 - Artificial image data generator.

```
plt.imshow(image_gen.random_transform(train_image_1))
```

```
<matplotlib.image.AxesImage at 0x7f92f82e5f40>
```

Figure 3.12 - Manipulated image by data generator.

```
for i, image_class in enumerate(([dir for dir in os.listdir(train_path) if dir != ".DS_Store"])):
    data = image_gen.flow_from_directory(
        train_path,
        target_size = image_size,
        color_mode = 'rgb',
        class_mode = 'binary',
        classes = [image_class],
        save_to_dir=train_augmented_path +'/' + image_class,
        save_prefix='augmented_',
        save_format='png',
        batch_size=32)
    data.next()
```

```
Found 1 images belonging to 1 classes.
Found 1 images belonging to 1 classes.
Found 2 images belonging to 1 classes.
```

Figure 3.13 - Augmented image generation for each category.

3.3.4 CNN for components classification

In order to use generated data set in the best way, was selected a model with includes 3 convolutional layers for image processing. The code presented in Figure 3.14.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dropout, Flatten, Dense, Conv2D, MaxPooling2D
```

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
model.add(MaxPooling2D(pool_size=(2, 2)))


model.add(Flatten())


model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Figure 3.14 - CNN for components classification.

The designed and trained model shows a 0.99 accuracy, which is a remarkable result for image classification tasks, Figure 3.15.

```
model.metrics_names
```

```
['loss', 'accuracy']
```

```
model.evaluate_generator(test_image_gen)
```

```
[0.020661674439907074, 0.9903846383094788]
```

Figure 3.15 - Model evaluation.

3.3.5 Prediction on an image

The model was verified on test data set files and showed correct categorization results. As an example a "Best Sellers" image, Figure 3.16., was correctly categorized by the model. Results on categorization are depicted in Figure 3.17.

Figure 3.16 - Image for categorization.

```
predict_image.predict(my_image)
```

```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

```
train_image_gen.class_indices
```

```
{'availability': 0,
 'best_sellers': 1,
 'computers': 2,
 'curbs-side_pickup': 3,
 'drones': 4,
 'free_shipping': 5,
 'low_prices': 6,
 'sale': 7,
 'shop_now': 8,
 'werable': 9}
```

Figure 3.17 - Categorization results.

3.3.6 Automated web application testing with neural network comparison

As a result of an extended test framework with functions for components extraction and comparison by neural network and trained convolutional neural network with augmented dataset, a complete end-to-end testing flow was created. A test framework runs a test script and on specific steps sends visual components for comparison and validation, as depicted on Figure 3.18.

Figure 3.18 - Categorization results.

Design and implemented prototype of the system shows the viability of the solution and its benefits over traditional testing approaches.

As further improvements the test framework and the neural network could be modified to be able to test and verify more sophisticated scenarios such as inter components relations or whole page recognition and verification.

Conclusions

The first section of this work includes an overview of test automation practice as a part of software development and its tools and techniques. Explained a need for automated testing and reasons why testing dramatically changed during the last decade. Overview of how testing in general and automated testing, in particular, affects software complexity, refactoring procedures and improves, in general, a development process for organizations. Added review for existing automated testing tools, their benefits, and issues. Highlighted and need of automation framework architecture and its contribution to solution architecture. Continuous integration and continuous testing are a part of this section as automation is an obligatory part of these two approaches. The section summary stated issues with traditional automation testing tools and demand for new generation tools.

The second section, dedicated to the design of automated testing tools with artificial intelligence components. Architecture documentation prepared by C4 + 1 modes for visualizing software architecture. The section includes a system context diagram, container diagram, component diagram for test framework, and components screenshot engine. As a summary added a Deployment diagram with an explanation of architecturally significant requirements, performance, scalability, stability, and maintainability addressed in the design.

The third section consists of the system prototype implementation. Here presented the steps of implementation of functional test extension, convolutional neural network (CNN) for component comparison. Described how to build a dataset and additional required steps with image transformation and mutation for successful CNN model train. During the work, a CNN model was trained and tested against real application components and showed satisfactory comparison results. Also implemented CNN model showed ~ 96%+ predictability results.

In a summary, implemented system design and prototype confirms the relevance and effectiveness of the automated testing with machine learning tools and cloud-based deployment could provide scalability and cost-efficiency to the approach. The design could be used as a prototype for further development of enterprise-grade testing tools.

References

1. Myers, G.: The Art of Software Testing. John Wiley & Sons, N.Y. (2004)
2. Axelrod A. (2018) Other Types of Automated Tests. In: Complete Guide to Test Automation. Apress, Berkeley, CA.
3. Graves, T., Harrold, M., Kim, J., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology (TOSEM) 10(2), 184-208 (2001).
4. Li, P., Huynh, T., Reformat, M., Miller, J.: A practical approach to testing GUI systems.
   Empirical Software Engineering 12(4), 331-357 (2007).
5. Kateryna Ivanova, Galyna Kondratenko, Ievgen Sidenko, Yuriy Kondratenko: Artificial Intelligence in Automated System for WebInterfaces Visual Testing. [Online]. Available: http://ceur-ws.org/Vol-2604/paper68.pdf
6. Kondratenko, Y., Gordienko, E.: Neural Networks for Adaptive Control System of Cater-
   pillar Turn. In: Annals of DAAAM for 2011 & Proceeding of the 22th Int. DAAAM Symp. "Intelligent Manufacturing and Automation", 2011, pp. 0305-0306.
7. Lönnberg, J.: Visual testing of software. Master's thesis, Helsinki University of Technology (2003).
8. Weyuker, E.: Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering 1215(12), 1128-1138 (1986).
9. Schneider, G., Winters, J.: Applying Use Cases: A Practical Guide. Addison Wesley, Boston (1998).
10. Pomanysochka, Y., Kondratenko, Y., Kondratenko, G., Sidenko, I.: Soft computing techniques for noise filtration in the image recognition processes. In: IEEE 2nd Ukraine Conference on Electrical and Computer Engineering, UKRCON, pp. 1189-1195, Lviv, Ukraine (2019).

11. Cewu Lu, Ranjay Krishna, Michael Bernstein, Li Fei-Fei: Visual Relationship Detection with Language Priors. [Online]. Available: https://arxiv.org/pdf/1608.00187.pdf
12. Mohammad Amin Sadeghi1,2 , Ali Farhadi: Visual Phrases. [Online]. Available: http://vision.cs.uiuc.edu/phrasal/recognition_using_visual_phrases.pdf
13. Bo Dai, Yuqi Zhang, Dahua Lin: Deep Relational Network. [Online]. Available: https://arxiv.org/pdf/1704.03114.pdf
14. Mohanty, H., Mohanty, J., Balakrishnan, A.: Trends in Software Testing. Springer, Singapore (2017).
15. Robert, M., Linda, H., Shapiro, G.: Image segmentation techniques. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0734189X85901537
16. Kaur, D., Kaur, Y.: Various Image Segmentation. Techniques: A Review 3(5), 809-814 (2014).
17. Hlaing, S., Khaing, A.: Weed and crop segmentation and classification using area thresholding. International Journal of Research in Engineering and Technology, 2321-7308 (2014).
18. Lottes, P., Stachniss, C.: Semi-Supervised Online Visual Crop and Weed Classification in Precision Farming Exploiting Plant Arrangement. [Online]. Available: http://flourishproject.eu/fileadmin/user_upload/publications/lottes17iros.pdf.
19. Deep Visual-Semantic Alignments for Generating Image Descriptions. [Online]. Available: https://cs.stanford.edu/people/karpathy/deepimagesent/
20. Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan: Show and Tell: A Neural Image Caption Generator. [Online]. Available: https://arxiv.org/pdf/1411.4555.pdf

21. Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio: Neural Machine Translation by Jointly Learning to Align and Translate. [Online]. Available: https://arxiv.org/pdf/1409.0473.pdf
22. Image Captioning. [Online]. Available: http://shikib.com/captioning.html
23. Kushneryk, P., Kondratenko, Y., Sidenko, I.: Intelligent dialogue system based on deep learning technology. In: 15th International Conference on ICT in Education, Research, and Industrial Applications: PhD Symposium (ICTERI 2019: PhD Symposium), vol. 2403, pp. 53-62, Kherson, Ukraine (2019). http://icteri.org/icteri-2020/PhDS/11110053.pdf.
24. Siriak, R., Skarga-Bandurova I., Boltov, Y.: Deep Convolutional Network with Long Short-Term Memory Layers for Dynamic Gesture Recognition. In: 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), pp. 158-162, Metz, France (2019). DOI: 10.1109/IDAACS.2019.8924381.
25. Wang, Y., Chen, Y., Lu, P., Wang, H.: Sobel Heuristic Kernel for Aerial Semantic Segmentation. In: 25th IEEE International Conference on Image Processing (ICIP), Electronic ISSN: 2381-8549.
26. Fan, W.: Color image segmentation algorithm based on region growth. Computer Engineering 36(13), 25-34 (2010).
27. Angelina, S., Suresh, L., Veni, S.: Image segmentation based on genetic algorithm for region growth and region merging. In: International Conference on Computing, Electronics and Electrical Technologies (ICCEET), pp. 970-974 (2012).
28. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/
29. Simon Brown: The C4 model for visualising software architecture. [Online]. Available: https://c4model.com/
30. Mario Carrion: Using the C4 model to document software architecture. [Online]. Available:

https://mariocarrion.com/2020/12/30/documenting-software-architecture-c4-model.html

31. Cypress. [Online]. Available: https://www.cypress.io/
32. Aurélien Géron: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, Inc., 2019.

Abbreviation and terms

UI - user interface.

GUI - graphical user interface.

UX - user experience.

ML - machine learning.

SUT - the system under test.

CNN (convolutional neural network) - a machine learning technique, one of the classes of deep neural networks. Frequently applied to analyze visual images.

CI - continuous integration.

CD - continuous delivery.