information

Article

An Image-Based User Interface Testing Method for Flutter Programming Learning Assistant System

Soe Thandar Aung *, Nobuo Funabiki *, Lynn Htet Aung, Safira Adine Kinari, Khaing Hsu Wai and Mustika Mentari

Department of Information and Communication Systems, Okayama University, Okayama 700-8530, Japan
* Correspondence: soethandar@s.okayama-u.ac.jp (S.T.A.); funabiki@okayama-u.ac.jp (N.F.)

Abstract: *Flutter* has become popular for providing a uniform development environment for user interfaces (UIs) on smart phones, web browsers, and desktop applications. We have developed the *Flutter programming learning assistant system (FPLAS)* to assist its novice students' self-study. We implemented the *Docker-based Flutter environment* with *Visual Studio Code* and three introductory exercise projects. However, the correctness of students' answers is manually checked, although automatic checking is necessary to reduce teachers' workload and provide quick responses to students. This paper presents an *image-based user interface (UI) testing method* to automate UI testing by the answer code using the *Flask* framework. This method produces the UI image by running the answer code and compares it with the image made by the model code for the assignment using *ORB* and *SIFT* algorithms in the *OpenCV* library. One notable aspect is the necessity to capture multiple UI screenshots through page transitions by user input actions for the accurate detection of changes in *UI elements*. For evaluations, we assigned five *Flutter* exercise projects to fourth-year bachelor and first-year master engineering students at Okayama University, Japan, and applied the proposed method to their answers. The results confirm the effectiveness of the proposal.

Keywords: Flutter; FPLAS; testing; image; Flask; OpenCV; user interface

check for updates

Citation: Aung, S.T.; Funabiki, N.; Aung, L.H.; Kinari, S.A.; Wai, K.H.; Mentari, M. An Image-Based User Interface Testing Method for Flutter Programming Learning Assistant System. *Information* **2024**, *15*, 464. https://doi.org/10.3390/info15080464

Academic Editors: Ricardo Queirós and Mário Pinto

Received: 15 June 2024
Revised: 27 July 2024
Accepted: 31 July 2024
Published: 3 August 2024

1. Introduction

Recently, integrating *mobile app* developments into educational curricula [1] has become increasingly crucial due to the ubiquitous presence of mobile devices and the growing demand for digital skills in the modern workforce. As smartphones and tablets become integral in everyday life, understanding how to create and manage mobile applications is essential for university students aiming to thrive in technology-driven environments. By incorporating mobile app developments into their teaching programs, educational institutions can equip students with practical, industry-relevant skills highly valued in today's job market [2].

Flutter [3] is a prominent framework developed by *Google* that utilizes the *Dart* programming language [4]. This combination offers a robust and flexible solution for building cross-platform applications. *Flutter* enables developers to write a single codebase that can be deployed on multiple platforms, including *iOS*, *Android*, *web browser*, and *desktop*, significantly reducing development time and effort. One of *Flutter's* standout features is its "hot reload" capability, which allows developers to instantly see the results of their code changes without restarting the application. This feature is particularly beneficial in an educational setting, as it fosters a more interactive and iterative learning process, enabling students to experiment and see the immediate impacts of their work.

Adopting *Flutter* in educational curricula can enhance the learning experience by providing students with hands-on opportunities to engage with current technologies. It also aligns with the latest trends in software development, ensuring that the skills acquired by students are relevant and up-to-date. Additionally, using a single, unified framework for multiple platforms helps streamline the learning process, allowing students to focus on mastering core programming concepts and application design principles without being overwhelmed by platform-specific complexities.

To support the independent learning of *Flutter* and *mobile app* development for novice students, we have developed the *Flutter programming learning assistant system (FPLAS)* [5]. *FPLAS* is designed to provide an accessible and streamlined entry point for beginners, integrating a *Docker*-based *Flutter* development environment. This setup ensures that novice students can easily access a consistent and pre-configured development environment via *Visual Studio Code* [6], regardless of whether they use *Windows*, *Linux*, or *Mac*. By eliminating the often cumbersome manual configuration processes, *FPLAS* allows students to focus directly on learning and developing their *mobile app* projects. To facilitate hands-on learning, *FPLAS* includes three sample projects that guide students through essential concepts and techniques in *Flutter* development.

Upon completing their projects, students must submit their source codes to the teacher for evaluation. This assessment process involves the manual execution of the source code and the manual inspection of the *user interface (UI)* output by the code. Since the teacher must repeat the process for every source code from many students, it is very time-consuming. Moreover, this manual checking of multiple projects in large classes can be particularly challenging for the teacher, potentially leading to delays or mistakes in feedback and grading. As the demand for digital skills education grows, developing more efficient assessment methods in *FPLAS* will be crucial to support educators and students, ensuring timely and compelling learning experiences.

This paper presents an *image-based user interface (UI) testing method* designed to automate the testing of UIs generated by students' answer codes using the *Flask* framework. This method involves executing the answer code on *Flask*, capturing the UI image by the code, and comparing this image with the corresponding UI image generated by the model code for the assignment. To perform this comparison, we employ the *ORB (Oriented FAST and Rotated BRIEF)* and *SIFT (Scale-Invariant Feature Transform)* algorithms available in the *OpenCV* library. These algorithms are used to detect and describe local features in the images, allowing for a robust comparison that can identify similarities and differences with high precision. By automating the UI testing process, our method significantly reduces the work required in grading, enhances the accuracy of the assessment, and provides timely feedback to students.

For evaluations of the proposal, we assigned five *Flutter* exercise projects to primarily fourth-year bachelor and first-year master students of engineering at Okayama University in Japan and applied the proposed method to their submitted source codes. This study involved these students to evaluate the system's effectiveness in an educational setting. These exercise projects were designed to cover a range of fundamental concepts and techniques in *Flutter* development, ensuring a comprehensive assessment of both the students' understanding and the system's capabilities. The results of this evaluation confirmed the effectiveness of our proposed method, demonstrating that the *automated UI testing method* significantly reduces the time and effort required in manual grading while maintaining high accuracy in the assessment process.

The rest of this paper is organized as follows: Section 2 introduces related works in the literature. Section 3 introduces adopted open-source software. Section 4 reviews the *FPLAS* platform. Section 5 presents the *image-based UI testing method* for *FPLAS*. Section 6 evaluates the proposal through applications to answer codes from novice students in five *Flutter* projects. Section 7 concludes this paper with future works.

2. Literature Review

In this section, we review the literature relevant to the topics discussed in our paper. We organize the section into three main themes: *programming education*, *UI testing*, and *image detection algorithms*.

2.1. Programming Education

In [7], Khan et al. addressed the educational challenges in Pakistan and the global shift towards online learning due to COVID-19. Their study focuses on bridging the gap between tutees and tutors to enhance academic achievement and character development. They promote peer tutoring, highlighting its benefits for both tutees and tutors. Their study emphasizes the positive impact of peer tutoring on education in Pakistan and supports its integration into educational platforms, suggesting further research to enhance its effectiveness.

In [8], Boada et al. introduced a *web-based tool* to enhance introductory programming courses, benefiting both teachers and students. For teachers, it enhances traditional teaching methods by reinforcing lectures and laboratory sessions, enabling personalized student attention, assessing student participation, and conducting continuous progress assessments. The tool offers a learning framework with help and correction environments to students, facilitating their work and increasing motivation for programming.

In [9], Crow et al. analyzed computer programming education, mainly the systems developed for tertiary courses. While these systems address difficulties faced by novices, they vary widely in design and offer diverse supplementary features such as interactive exercises, plans, quizzes, and worked solutions. This review highlights the need to support a broader range of supplementary features in intelligent programming tutors to enhance their effectiveness.

In [10], Keuning et al. conducted a systematic literature review on feedback in programming exercise tools, examining feedback content, generation techniques, adaptability, and tool evaluations. They analyzed 101 tools, finding that feedback often focuses on identifying mistakes rather than fixing them or guiding students forward. Tools vary in their abilities to adapt to teachers' needs, with limited diversity in feedback types. However, newer tools show promising trends in offering more diverse feedback. Various techniques, including data-driven approaches, are used for feedback generation, though challenges remain in evaluating tool effectiveness and facilitating teacher adaptation.

2.2. UI Testing

In [11], Sun et al. discussed the challenges in *mobile application testing*, particularly in locating *UI* components on screenshots. They proposed an *app UI component recognition system* based on image understanding. By analyzing *Android UI* component information and using image understanding techniques, they extracted component images from screenshots. They designed and implemented a *convolutional neural network (CNN)* [12] to classify these images, achieving a classification accuracy of 96.97%. Their approach extracts component information from screenshots, offering new solutions for challenging application scenarios.

In [13], Khaliq et al. employed AI in software testing to automate test case generations directly from *UI* element descriptions, reducing reliance on manual extractions. They utilized object detection algorithms and text-generation transformers to translate *UI* descriptions into executable test scripts. Their examinations of 30 e-commerce applications showed a high accuracy rate in generated test cases (up to 98.08%) and a significant decrease in test flakiness (average reduction of 96.05%).

In [14], Wang et al. discussed the existing *Android UI* testing tools for industrial apps and introduced *TOLLER* as an infrastructure-enhanced solution. It directly accesses runtime memory, notably reducing time spent on *UI* operations compared to *UIAutomator* [15]. The integration of *TOLLER* enhances existing *UI* testing tools, leading to substantial improvements in code coverage and crash detection capabilities. These findings emphasize the importance of infrastructure support in advancing *Android* testing tools.

2.3. Image Detection Algorithms

In [16], Tareen et al. thoroughly compared *SIFT*, *SURF*, *KAZE*, *AKAZE*, *ORB*, and *BRISK* algorithms for feature-based image registration, emphasizing the scale, rotation, and viewpoint invariance. Experiments involve matching scaled, rotated, and perspectivetransformed images with originals from diverse datasets. *SIFT* and *BRISK* are the most accurate, while *ORB* and *BRISK* demonstrate the highest efficiency, aiming to establish a benchmark for vision-based applications.

In [17], Zhong et al. proposed an improved *SIFT* algorithm to enhance image feature point acquisition by addressing multi-scale variations, noise, light intensity, and rotation issues. They used a *Difference of Gaussians (DOG)* pyramid to identify feature points. They replaced traditional descriptor construction with the *BRISK* algorithm, which generates faster, more unique binary descriptors using concentric circles in uniform sampling. Their group matching method finds the shortest *Hamming* distance between images, significantly reducing matching time and improving efficiency.

In [18], Gupta et al. addressed object recognition accuracy for applications like image classifications and surveillance, focusing on hand-crafted features. They used *ORB* and *SIFT* descriptors, with *SIFT* being particularly effective for images with varying orientations and scales. The study employed the *Locality Preserving Projection (LPP)* algorithm to reduce the dimensionality of the image feature vector. Testing on an 8000-sample, 100-class dataset with *k-NN*, decision tree, and random forest classifiers showed precision rates of 69.8% with *ORB*, 76.9% with *SIFT*, and 85.6% when combined.

In [19], Andrianova et al. introduced a four-stage approach for matching medical images using *SIFT* and *ORB* algorithms. Initially, key points and descriptors were identified, followed by clustering to determine optimal clusters and exclude outliers. "Good" matches were identified through *Lowe's* ratio test and the homography method, with noise rejected. A practical medical diagnostics example validated the effectiveness of this method.

In [20], Chhabra et al. introduced a *content-based image retrieval (CBIR)* system utilizing *ORB* and *SIFT* feature descriptors. They applied *K-means* clustering and *Locality-preserving* projection for dimensionality reduction. The evaluation of the Wang and Corel databases demonstrated a precision rate of 99.53% and 86.20%, respectively, when combining *ORB* and *SIFT* feature vectors.

3. Adopted Software Tools

This section introduces the *software tools* adopted in this paper for completeness and readability. For implementation details, we specifically describe the adopted software and their versions as follows: *Flask* (3.0.2), *Python* (3.11.8), *Flutter* (3.22.0), and *OpenCV* (4.9.0).

3.1. Flask

Flask [21] is a lightweight and flexible web framework for *Python* designed to simplify the development of web applications with minimal complexity. Its modular design allows developers to select the necessary components and easily extend functionality with third-party libraries. Despite its simplicity, *Flask* supports essential features such as routing, request handling, and templating, making it suitable for small-scale projects and large web applications. Its ease of use and straightforward approach make it a popular choice for beginners, while its scalability and the ability to customize make it favored by experienced developers. *Flask's* active community and extensive documentation further enhance its appeal, ensuring developers have the resources and support to build robust web applications efficiently.

3.2. OpenCV

OpenCV (Open Source Computer Vision Library) [22] is an open-source software library that specializes in computer vision and machine learning. Designed to provide a common infrastructure for computer vision applications, it supports a wide range of functionalities, including image processing, object detection, facial recognition, and motion tracking. *OpenCV* is written in C++ and has interfaces for *Python*, *Java*, and *MATLAB* , making it accessible to a broad audience of developers and researchers. Its comprehensive tools and algorithms enable the fast development of real-time vision applications, from fundamental image transformations to complex video analytics. Widely used in academia and industry, *OpenCV* benefits from a strong community that contributes to its extensive documentation and continuous improvement, ensuring it remains at the forefront of computer vision technology.

3.2.1. *ORB*

ORB (Oriented FAST and Rotated BRIEF) [23] is a robust feature detector and descriptor used in computer vision for tasks such as image matching, object recognition, and 3D reconstruction. Developed to provide a fast and efficient alternative to the well-known *SIFT* and *SURF* algorithms, *ORB* combines the *FAST* keypoint detector and the *BRIEF* descriptor while adding rotation invariance and noise resistance. It excels in real-time applications due to its computational efficiency and has been widely adopted in various fields, including robotics and augmented reality (AR). *ORB's* ability to maintain high performance even under varying lighting conditions and perspectives makes it a popular choice for feature extraction in diverse computer vision applications.

3.2.2. *SIFT*

SIFT (Scale-Invariant Feature Transform) [24] is a powerful and widely used algorithm in computer vision for detecting and describing local features in images. *SIFT* identifies distinctive key points and generates invariant descriptors for scale and rotation, and partially invariant descriptors for affine transformations and illumination changes. This robustness makes *SIFT* highly effective for image stitching, object recognition, and 3D reconstruction tasks. By extracting stable and reliable features, *SIFT* enables accurate matching between different views of the same scene or object, facilitating various applications in academic research and industry. Despite being computationally intensive, its precision and reliability have made it a cornerstone in feature extraction and matching.

3.3. GitHub

GitHub [25], a web-based platform for version control, plays a pivotal role for developers globally. Utilizing *Git*, an open-source version control system, *GitHub* facilitates hosting both open-source and private repositories. Its comprehensive suite of tools enables collaborations, allowing developers to efficiently manage and track changes in their code bases. Through features like pull requests, issues, and *wikis*, *GitHub* encourages a collaborative environment where users can host, review, and manage projects effortlessly.

3.4. Moodle

Moodle [26] is a widely used open-source *learning management system (LMS)* that facilitates online learning and course management. With its user-friendly interface and extensive features, *Moodle* allows educators to create dynamic online courses, manage content, track student progress, and facilitate communication and collaboration. Its modular architecture and plugin system enable customization to suit various educational needs and preferences. Used by academic institutions, businesses, and organizations worldwide, *Moodle* provides a flexible and scalable platform for delivering engaging and interactive online learning experiences.

4. Review of FPLAS Platform

This section reviews the *FPLAS* platform developed in our previous work [5].

4.1. Overview of FPLAS

FPLAS is a *Docker*-based *Flutter* development environment system designed for novice students to initiate mobile application development while avoiding the complexities of environment setups. Figure 1 shows the overview. It includes preparing a *Docker container image* with *system startup files* on *Docker Hub*. Additionally, two system startup files for different operating systems were added to *GitHub*. Sample *Flutter* projects are provided as code modification exercises to guide students in this environment. Additionally, comprehensive instructions on *GitHub* were provided to assist students in effectively utilizing the system, including steps for installing *Docker* [27] and *VSCode* [28], downloading the *Docker* image, connecting to the container, accessing the exercise projects, and submitting answer files to the teacher.

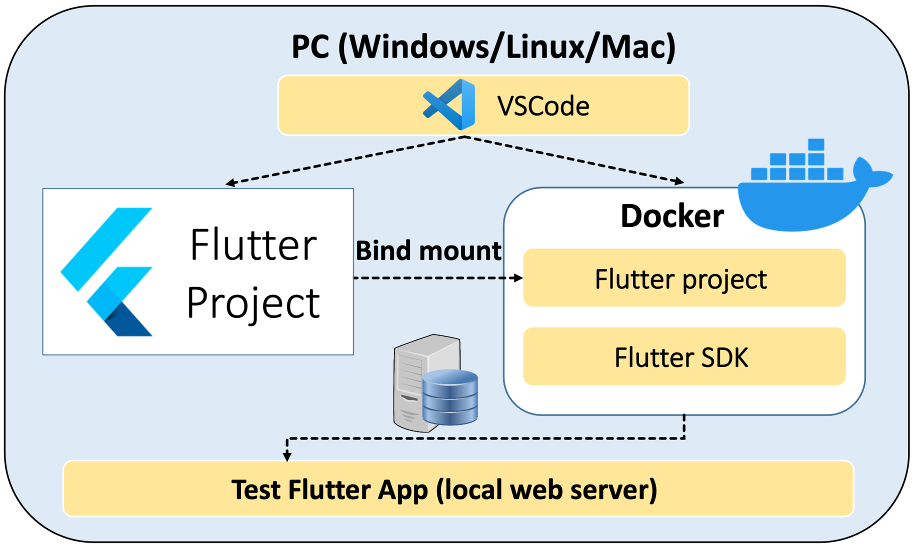


Figure 1. Overview of *Docker*-based *Flutter* development environment.

4.2. Usage Procedure and Access Exercises by Student

To install and initiate the *Flutter* project environment in the *Docker* container, students need to follow these steps, with instructions provided on *GitHub*:

1. Install *Docker* and *VSCode* based on the student's PC operating system.
2. Import the three extensions for *Flutter*, *Remote Development*, and *Docker* in *VSCode*.
3. Obtain the *Docker* container image for *FPLAS*.
4. Download the *GitHub* project containing the essential files, or clone the project if students have already installed *Git* on their PCs.
5. Open the downloaded project in *VSCode*, initiate the containerized development to access the remote development, and activate the *FPLAS* development environment in the *Docker* container.

After connecting to the *FPLAS* development environment in the *Docker* container, students can start solving the three exercises included in the container by following the steps below:

1. Transfer each exercise to the designated workspace in the container.
2. Access to the exercise directory. This allows students to navigate to the specific exercise folder, where they can modify the source codes according to the provided modification guidance.
3. Initiate the *Flutter* web server by executing *"flutter run -d web-server"*.
4. Preview the output generated by the source code by navigating to the local web server address using "http://localhost:port" in the web browser.

After completing the modifications for each exercise, students must submit the modified source code files through the provided folder on *Moodle*. Then, the teacher must manually assess their code files by executing them and checking their output *user interfaces (UIs)* individually.

5. Proposal of Image-Based UI Testing Method

In this section, we present the *image-based UI testing method* for *FPLAS*.

5.1. Software Architecture

Figure 2 shows the overview of the proposed *image-based UI testing method*. Firstly, this method utilizes *Flask*, a *Python* web framework, to create a user-friendly interface for selecting exercises and analyzing corresponding UI designs. The process retrieves the exercise and answer images from designated folders upon user selection.

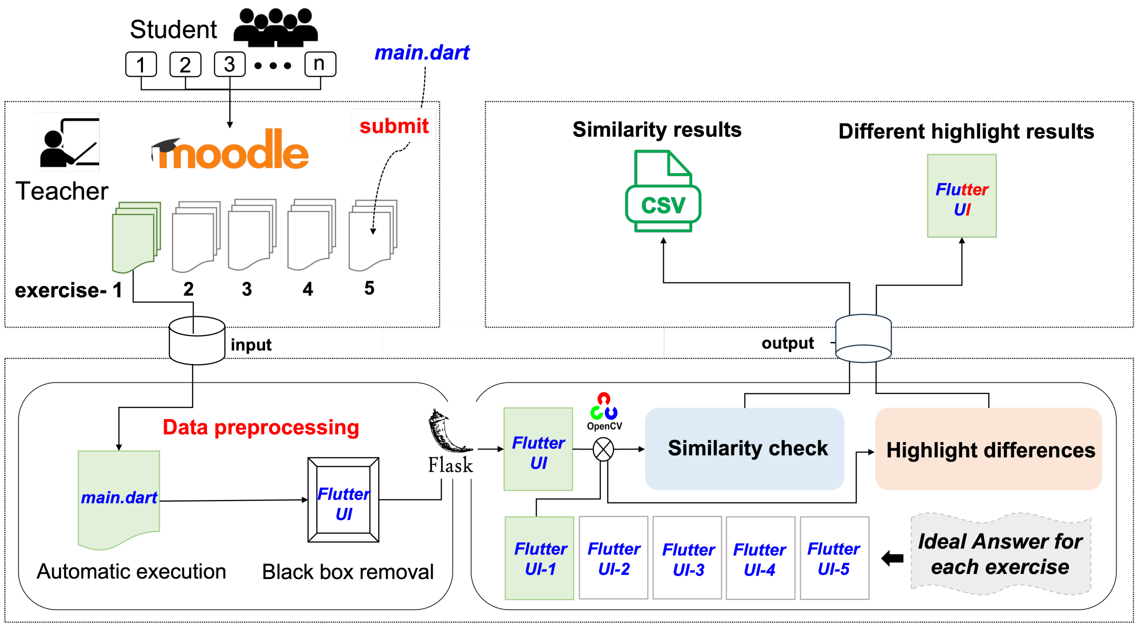


Figure 2. Overview of image-based *UI* testing method.

Next, the method employs image processing techniques to calculate the similarity between expected and actual *UI images*. This involves resizing images to a standard resolution, detecting key points, computing descriptors, and matching features, using the *SIFT (Scale-Invariant Feature Transform)* algorithm for images of the exact resolution and *ORB (Oriented FAST and Rotated BRIEF)* algorithm for images of different resolutions.

Different image resolutions were considered because the proposed method might be used by various teachers using different PCs with other specifications. Consequently, the resolution of the images can vary. To accommodate such situations, *SIFT* is used for images of the same size, and *ORB* is used for images that need resizing. Additionally, the method highlights differences between images by generating overlays that accentuate discrepancies, aiding in quick identification and analysis.

5.2. Data Pre-Processing for UI Testing

Before starting the process of the *UI testing method*, we need to collect the **"main.dart"** answer code files submitted by the students and preprocess them. This preprocessing consists of the following two steps.

5.2.1. UI Image Capture Step

To automate the UI image screenshot capture, this step needs to define the paths to the directories for the necessary resources: (1) the directory containing the individual **"main.dart"** files for each *Flutter* project, (2) the directory for the template *Flutter* project, and (3) the directory where the screenshots will be saved. Then, it utilizes *Python* libraries such as *os* [29], *shutil* [30], *subprocess* [31], *time*, *xdotool* [32], and *pyautogui* [33] to handle path manipulation and file operations, execute terminal commands, manage delays, and capture screenshots automatically.

First, this step copies the **"main.dart"** file into the template project directory. Second, it runs the *Flutter* application using the source code. This *Flutter* application is launched using the command **"flutter run -d chrome"**, which directs *Flutter* to run the application in a *Chrome* browser window. This process waits for the application to be fully loaded and searches for the specific *Chrome* window where the application is running. Once found, the window is activated and maximized to ensure the entire application interface is visible.

Third, it captures the window's geometry and uses this information to take a screenshot of the specified region. Fourth, it saves the screenshot in the predefined directory. Finally, it terminates the *Flutter* process.

By iterating over each **"main.dart"** file, the step extracts the student ID from the filename to uniquely identify and save the corresponding screenshot. It is noted that each filename contains the student ID. Each screenshot is saved with the filename that corresponds to the student ID, facilitating easy identification and further analysis. This preprocessing step is essential in the image-based UI testing method, as it ensures that all necessary *UI* images are captured accurately and consistently across multiple applications, providing a reliable dataset for subsequent image-based analysis and validation.

5.2.2. Black Border Removal Step

After capturing the screenshots as part of the UI testing process, the next preprocessing step involves refining these images to ensure accuracy in the subsequent analysis. Here, any screenshot image contains a black border created by the *Flutter* code in the provided exercise project. The unnecessary black borders that may appear around the UI elements are eliminated in the screenshots.

To automate this step, we utilize the *OpenCV* library, systematically processing each image in the designated input folder. By converting the images to grayscale and identifying the most prominent contour, the step accurately determines the bounding box of the UI elements. The region within this bounding box is then cropped to produce a refined image free from excess borders. This step is essential to analyze the UI components by eliminating any background noise that could distort the results. The cropped images are saved in the specified output folder, which will serve as the input parameter in the next step, ensuring a structured and organized workflow. This preprocessing method enhances the accuracy of the image-based UI testing method by providing precise and uniform images for further evaluation.

5.3. Image-Based UI Testing

After the data pre-processing, we implemented the *UI testing method* using *Flask*. *OpenCV* is used for advanced image processing and manipulations, *PIL (Python Imaging Library)* [34] is used for handling image data, *numpy* [35] is used for numerical operations, and *pytesseract* [36] is used for *optical character recognition (OCR)*. The implemented method performs the following procedures to compare the expected UI image with the generated UI images by students' source codes and to highlight differences by integrating the image comparison algorithms, such as *SIFT* and *ORB*.

- **Directory Path:** The paths to specific directories are defined for hosting the images and results. They include the *"exercise"* directory for the correct images of the exercises, the *"answer"* directory for the corresponding answer images received from the preprocessing steps, the *"result"* directory for the processed results, and the *"difference"* directory for the images that highlight differences between correct images and answer images. These paths can ensure efficient file management and operations.
- **Image Size and Similarity Check:** The *check_image_size_similarity* function ensures that both correct and answer images have the exact matching dimensions. It calculates similarity percentages using either the *SIFT* or *ORB* algorithm, depending on whether the size is the same. Both methods detect vital key points and compute descriptors to match features between images, ultimately providing a similarity score that quantifies how closely the images match.
- **Similarity Calculation by SIFT:** *SIFT* is used when images have the same size, providing robust feature matching. It detects key points and computes descriptors in both images, and then matches them using a *FLANN*-based matcher [37]. Good matches are filtered to calculate the similarity percentage, offering an accurate measurement based on key point matching.
- **Similarity Calculation by ORB:** *ORB* is used when images need resizing, offering a faster alternative with lower computational complexity. The *resize_image* function

standardizes image dimensions, ensuring consistency and enhancing similarity calculations. *ORB* detects vital key points and computes descriptors, matches them using *BFMatcher* [38], and calculates similarity based on good matches, providing an efficient method for image comparison.

- **Image Difference Highlighting:** The results are sorted and saved in a *CSV* file, providing comprehensive analysis of image similarities. Then, to identify and highlight the differences in the images, the *highlight_image_difference* function computes the absolute difference between the images, applies a threshold to create a binary mask, and dilates this mask to enhance visibility. The differences are highlighted in red on the original image, and the result is saved for the user or teacher review.

Web Interface

The web interface sets up routes to handle HTTP requests and enable user interaction with the application. The '/' route is the main page, allowing users to select the exercise and submit its UI images. Upon submission, the application process contains the following procedures:

- **Image Retrieval:** Upon receiving the request, the application fetches the selected correct and answer images from the specified folders and prepares the paths for storing and accessing the result files.
- **Comparison Process:** The application checks whether the CSV file containing the similarity results exists for the selected exercise, and if not, it computes the image similarities using the defined functions and saves the results in a CSV file for future reference.
- **Data Presentation:** Once the comparison process is completed, the web interface displays the reference and student answer images for the selected exercise and provides an option to view detailed similarity results through a downloadable *CSV* file.

Finally, the *Flask* application runs the development server to host the web interface, allowing users to access the functionality through a web browser. Figure 3 shows the highlighting differences with similarity results in *UI* testing.

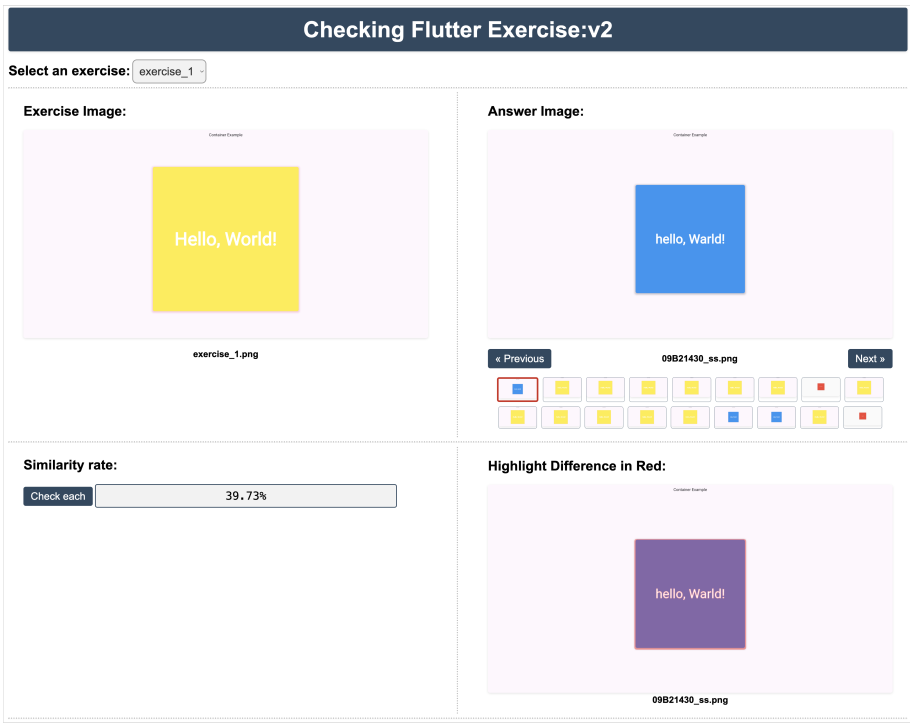


Figure 3. Highlighting differences with similarity result in *UI* testing.

6. Evaluation

In this section, we evaluate the proposed *image-based UI testing method* for *FPLAS* through applications to answer codes from novice students in five *Flutter* exercise projects.

6.1. Participant Overview and Methodology

The participants in this study were drawn from two engineering classes at Okayama University. One class consisted of 24 students who were assigned three programming exercises, while the other class consisted of 20 students who were assigned five programming exercises. The gender distribution in both classes was predominantly male (80%), with a minority of female students (20%). Despite their diverse academic backgrounds, none had prior experience with *Flutter*, though they were familiar with programming languages such as C, C++, and Java.

Our evaluation aimed to observe how these students adapted to learning mobile application developments using *Flutter*. With the support of a teaching assistant, the students set up the environment and completed the exercises either in the classroom or at home, submitting their answer code via *Moodle*. The research methodology involved a structured series of exercises and projects conducted in three weeks. Detailed procedures included the selection criteria for participants, the instructional design, and the data collection and analysis methods.

Participation in the study was mandatory, meaning that the students were required to engage in the study activities as part of the course requirements. This ensured full involvement from the whole class, providing a complete set of data for analysis. Consequently, the findings represented the entire class, which is critical for the validity of the study's outcomes. It also meant that all the students had to interact with the *Flutter Programming Learning Assistant System (FPLAS)*, ensuring consistent feedback on its usability and effectiveness.

Beyond the gender distribution, other demographic characteristics such as the age range, prior experiences with programming, and familiarity with *Flutter* could provide deeper insights into the study's context. For instance, the age range of participants was between 20 and 25 years. Most students had prior programming experiences but varied in their familiarity with programming.

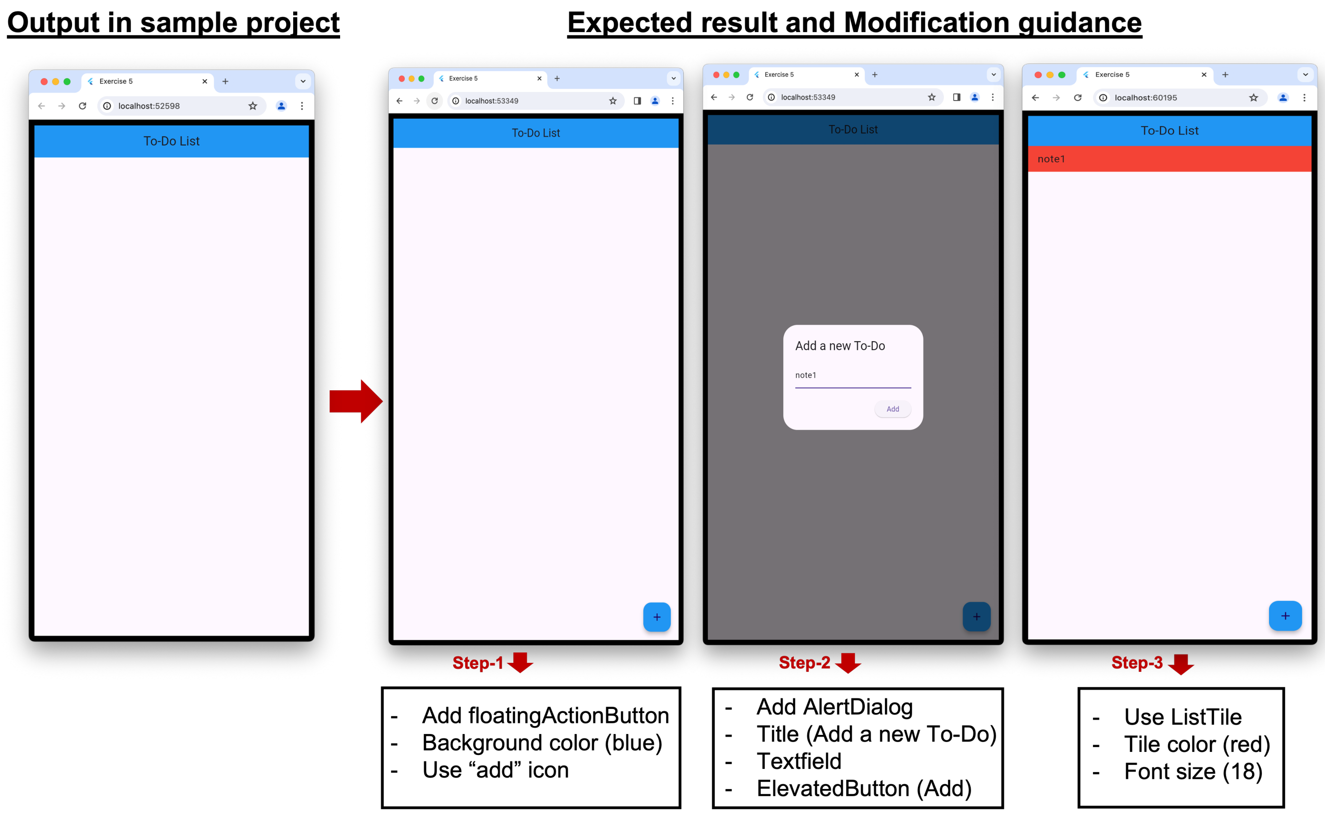
This study was conducted over two to three weeks. Initially, students were introduced to the *Flutter Programming Learning Assistant System (FPLAS)* and instructed on how to use it. They were then assigned the exercises. Each exercise was designed to cover specific programming skills and the use of *Flutter*. The teaching assistant supported the exercises, ensuring students could effectively use the system and resolve technical issues.

6.2. Five Flutter Projects for Exercises

The five *Flutter* projects are prepared for novice students aiming to initiate mobile application development in *Dart* programming. They provide basic learning experiences, covering essential *Flutter* widgets and components crucial for creating dynamic and interactive user interfaces. Each project offers hands-on practice and modification guidance. These exercise projects include: (1) *Dynamic UIs*: Widgets and State; (2) *App Structure*: Material and Scaffold; (3) *App Navigation*: AppBar and BottomNavigationBar; (4) *UI Styling*: Container, Text, and Buttons; (5) *User Interactions*: FloatingActionButton, TextFields, and Dialogs. Table 1 lists the objective and essential items as the guidance for the successful completion of the five exercise projects. Figure 4 shows the UI interfaces for creating a *Simple To-do List* application for **Exercise-5**.

Table 1. Five projects and their modification guidance.

Exercise	Objective	Modification Guidance
Exercise-1	Container widget as a fundamental UI element used to encapsulate other widgets.	• box size (400 × 400), box color (yellow) • box shade (pink), box text (50)
Exercise-2	ListView displays scrollable lists of widgets and manipulates their functionality.	• show the list in descending order • modify arrow direction
Exercise-3	AlertDialog widget for displaying critical information and interacting with users.	• icon color (red), button text, button style (outlined)
Exercise-4	BottomNavigationBar, layout widgets, text input, conditional UI updates, and asset management.	• maintain Page 1 and its original logic • Add Page 2 with similar logic to check for the word "cat" • Both pages have their respective text fields and check buttons
Exercise-5	Create a simple to-do list app with custom widget and state management, input dialog, list display and management, item addition, basic layout, and styling.	• Add floatingActionButton • Background color (blue) • Use the "add" icon text, Add AlertDialog, • Title, Textfield, ElevatedButton (Add) • Use ListTile, Tile color (red), Font size (18)



- Add floatingActionButton
- Background color (blue)
- Use "add" icon

- Add AlertDialog
- Title (Add a new To-Do)
- Textfield
- ElevatedButton (Add)

- Use ListTile
- Tile color (red)
- Font size (18)

Figure 4. *To-do list* project in **Exercise-5**.

6.3. Results of Five Exercises

Next, we discuss the application results of the proposed method to answer codes in five exercises from students. As shown in Figure 5, the similarity score results are depicted using bar graphs for all the images obtained from students' source codes. The color of each bar graph represents the result that is similar to the correct image. Here, *green* represents the answer image with a 100% similarity score, *blue* represents an intermediate score between 20% and 99%, and *pink* represents the image with a low score of less than 20%.

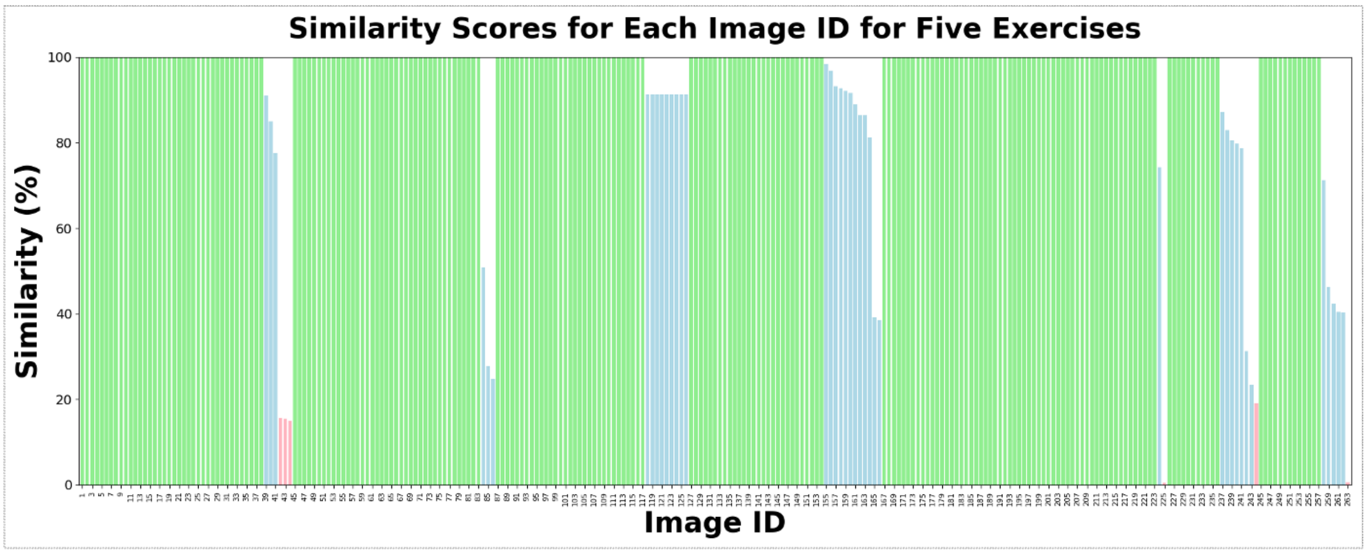


Figure 5. Similarity scores for all images of five exercises.

6.3.1. Results of Exercise-1 and Exercise-2

Figure 6 shows the application results for answer codes in **Exercise-1 (left)** and **Exercise-2 (right)**. In both exercises, the majority of images (**ID 1** to **ID 38** in **Exercise-1** and **ID 1** to **ID 39** in **Exercise-2**) exhibit the highest similarity scores of 100%, indicating the robust performance of the proposal. However, a noticeable decline in the score is observed from image **ID 39** onwards in **Exercise-1** and **ID 40** onwards in **Exercise-2**. The lowest scores were recorded for the last three images, highlighted by *pink* to indicate a low score of less than 20%. For reference, we show one *UI* image among them in Figure 6.

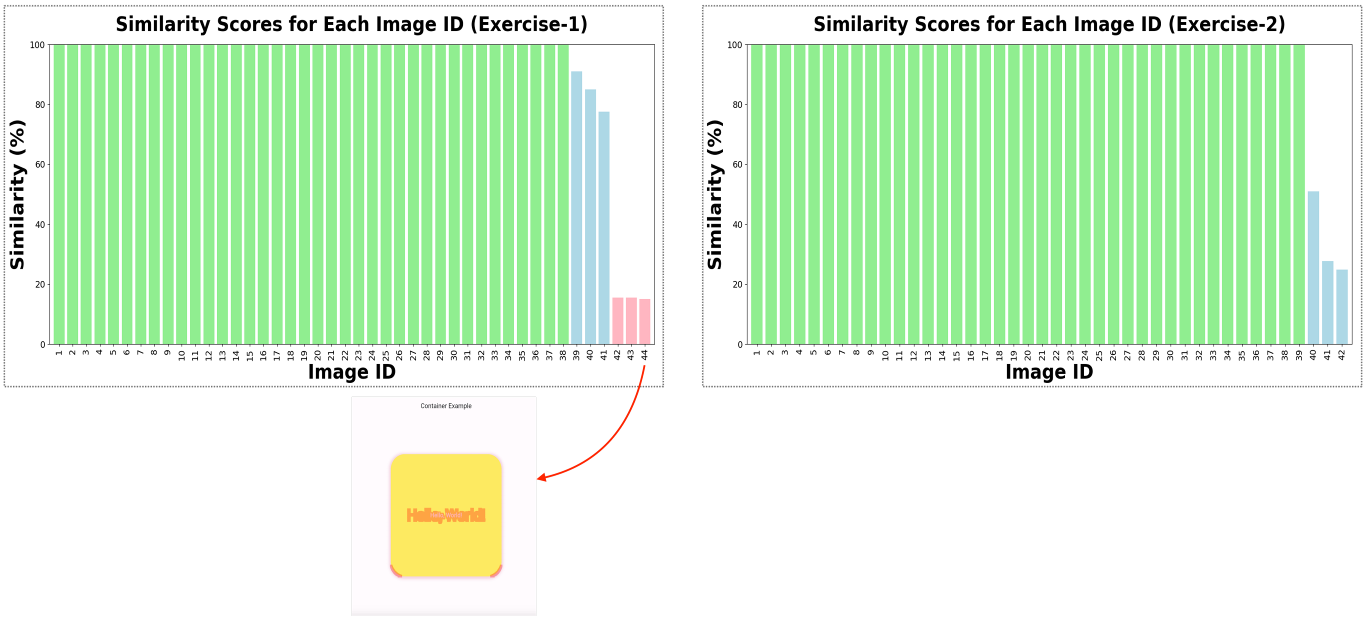


Figure 6. Results for **Exercise-1** and **Exercise-2**.

6.3.2. Result of Exercise-3

Exercise-3 requests to make two different user interfaces, labeled as **Exercise-3 (a)** and **Exercise-3 (b)**. Figure 7 shows the application results. For **Exercise-3 (a)**, the images with **IDs** from **1** to **31** have the highest similarity score of 100%, showing the consistent performance of the proposal. However, from **IDs 32** to **40**, the score drops to an intermediate level. For **Exercise-3 (b)**, the images with **IDs** from **1** to **28** also have 100% similarity scores. From **ID 29** onwards, the score gradually declines, although no low scores are less than 20%.

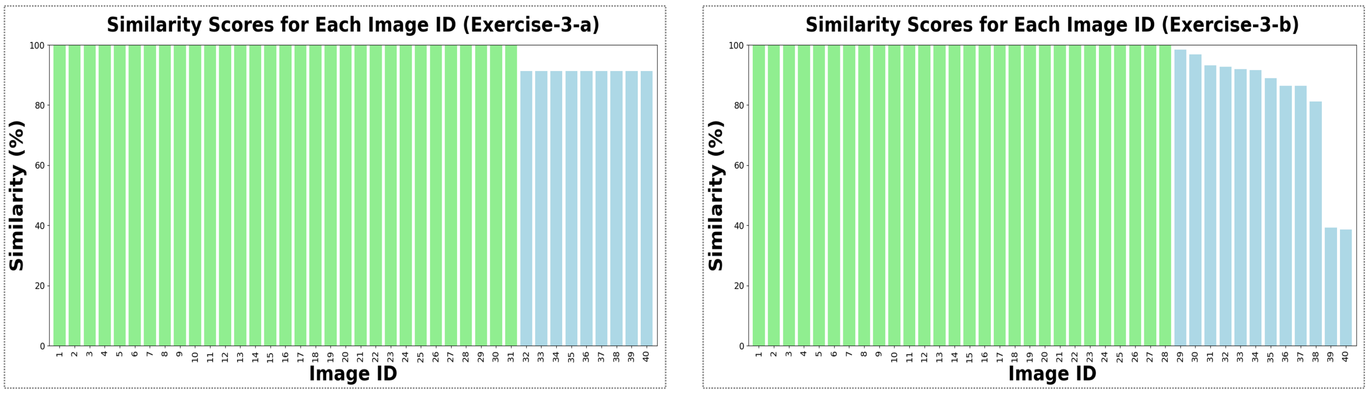


Figure 7. Results for **Exercise-3**.

6.3.3. Result of Exercise-4

Exercise-4 also requests to make two different user interfaces, labeled as **Exercise-4 (a)** and **Exercise-4 (b)**. Figure 8 shows the application results. In both graphs, every image has the highest similarity score of 100%. This consistent result suggests that the proposed image comparison method is highly effective, robust, and precise.

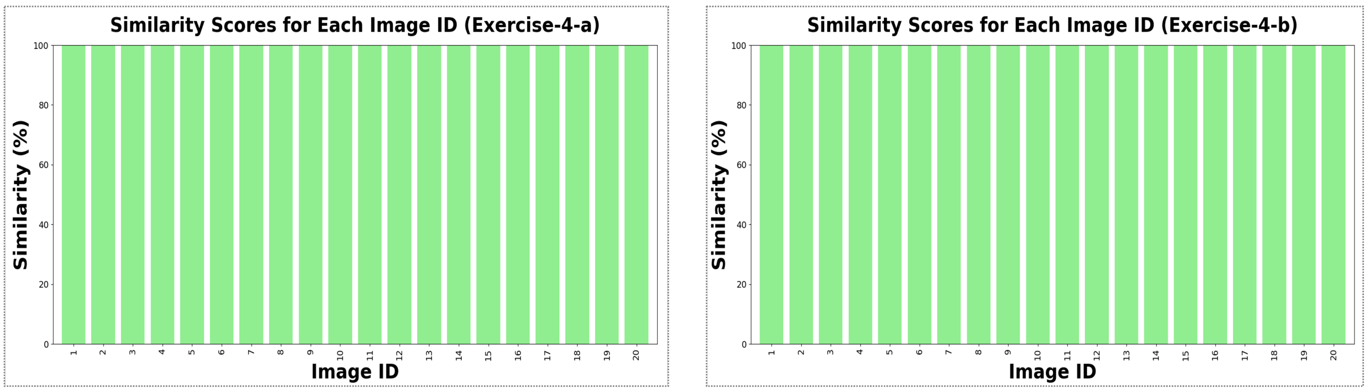


Figure 8. Results for **Exercise-4**.

6.3.4. Result of Exercise-5

Exercise-5 requests to make three different user interfaces, labeled as **Exercise-5 (a)**, **Exercise-5 (b)**, and **Exercise-5 (c)**. Figure 9 shows the application results. In **Exercise-5 (a)**, the first 17 images achieved 100% similarity score. However, the score of the last image was very low. In **Exercise-5 (b)**, the first 11 images achieved 100% similarity score. However, the scores of the last three images were low. In **Exercise-5 (c)**, the first 13 images achieved 100% similarity score. However, the score of the last image was very low. For reference, we show the corresponding image with the lowest score in each interface in Figure 9. We observed that the student for image ID 19 did not modify answer code. These results indicate that our proposed method works correctly and efficiently for various answer codes from students.

6.4. Discussion

In this section, we discuss the key findings of this study, the automation process, limitations, future works, and the advantages for university teachers.

6.4.1. Findings

The evaluation results of the *image-based UI testing method* using *Flask*, *ORB*, and *SIFT* algorithms have demonstrated significant advantages in automating the assessment of students' source codes for *Flutter* programming exercises. This method will provide an efficient and consistent means of assessing a lot of work from students. These findings are consistent with the research conducted by Muuli et al. [39], who also reported that the automated assessment using image recognition can significantly reduce grading workloads and provide timely feedback to students.

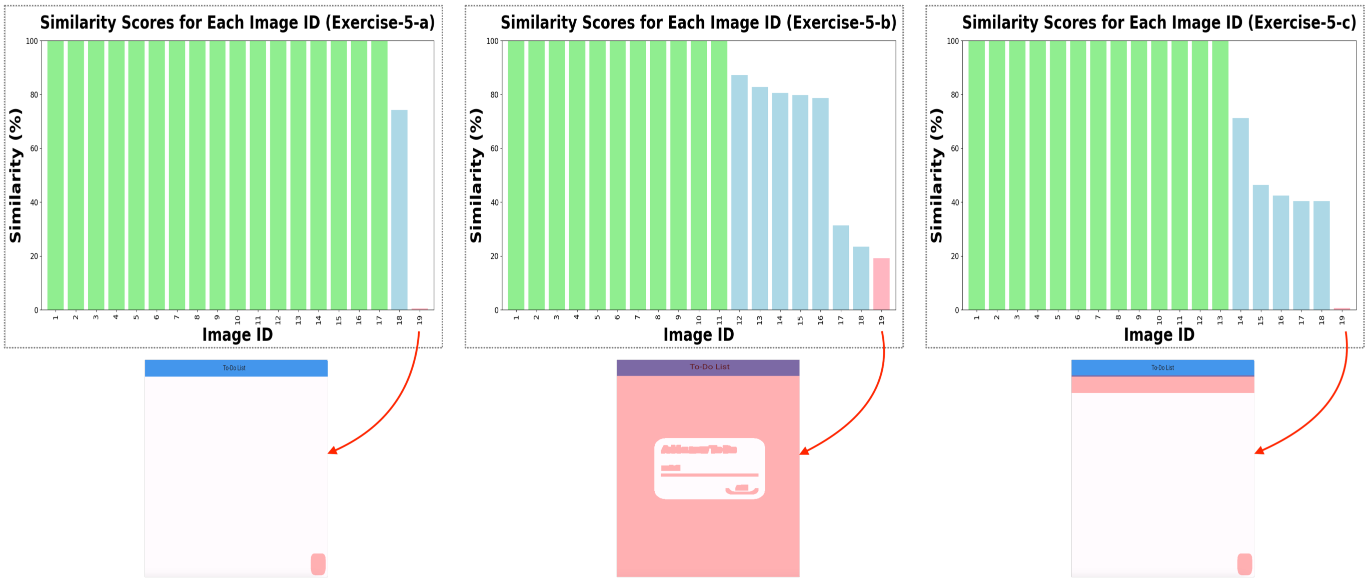


Figure 9. Results for **Exercise-5**.

Similar studies, such as those by Combefis et al. [40], have explored automated code assessment systems and their impact on educational settings. They found that automated systems can effectively handle large volumes of code submissions, providing quick and relevant feedback to learners. This aligns with our findings that the *Flutter Programming Learning Assistant System (FPLAS)* offers efficient and accurate feedback, enhancing the educational experience.

Another study by Mozgovoy et al. [41], which involved a case study of a mobile tennis game project developed in Unity, demonstrated the practical applicability of using the image-matching capabilities of the *OpenCV* library in Appium-supported functional tests. This approach built a reliable automated QA pipeline for nonstandard GUI applications like games. The study highlights the challenges and solutions of automated smoke testing in such contexts, which is relevant using the *image-based UI testing method* for *Flutter* applications.

Several commercial tools reflect the effectiveness of automated UI testing. For instance, *Rainforest QA* uses automated UI testing with features like the screenshot comparison and visual regression testing, which are essential for verifying UI changes and ensuring a consistent user experience [42]. Similarly, *Applitools Eyes* offers a robust solution for visual validation through its cloud-based image comparison API, integrating well with popular test automation frameworks like *Selenium* and *JUnit* [43]. *Screenster* combines visual regression testing with screenshot comparison to automate UI testing, supporting testing across multiple browsers and web applications [44].

However, there are some differences in the approaches and tools used. For instance, the study in [39] focused on image recognition for graphical outputs in programming tasks, whereas our approach integrates *multiple UI* elements and interactions specific to *Flutter* applications. This broader scope allows *FPLAS* to be more versatile and applicable to various programming exercises.

Our study contributes to the field by demonstrating the integration of multiple algorithms, such as *ORB* and *SIFT*, to automate the assessment of multiple UI interactions in *Flutter* applications. This approach offers a comprehensive solution for evaluating various aspects of student submissions, beyond simple code correctness. Additionally, the *FPLAS's* ability to provide quick and accurate feedback distinguishes it from other automated assessment tools, emphasizing its potential to improve learning outcomes.

In conclusion, our findings support the results of similar studies, underscoring the importance of automated assessment tools in modern education. By comparing our research with those of other authors, we highlighted the common benefits and unique contributions of *FPLAS*, setting the stage for further advancements in this field.

6.4.2. Automation of Multiple UI Screenshots

One notable aspect of our current implementation is the necessity to capture *multiple UI screenshots* through page transitions by user input actions for **Exercise-3**, **Exercise-4**, and **Exercise-5**. It is crucial to accurately detect the changes in *UI elements* caused by user actions. To enhance its automation, we have integrated functionalities for capturing screenshots upon detecting changes by user actions. The provided *UI image capture step* is designed to automate taking screenshots based on user input actions such as *button clicks* and *text field inputs*. While the script can calculate offsets from the window's bounds to trigger *UI* updates and document the resulting changes in the *UI*, the user input actions must be performed manually. After each input action, the script pauses to refresh the *UI* before taking subsequent screenshots.

6.4.3. Limitations and Future Work

A limitation of this study is the sample size, which consisted of 44 fourth-year bachelor or first-year engineering master students in Okayama University, Japan. While the results are promising, this relatively small sample size may limit the generalizability of the findings. Future research should involve a larger and more diverse group of participants to validate the effectiveness of *FPLAS* across different educational contexts.

Furthermore, the limitation of the current implementation is the manual user input actions for screen transitions. In future works, we will automate the user actions by exploring various approaches. One option is integrating *Flutter's* built-in testing framework using the *flutter driver* [45]. Another is using web scraping technology such as *Appium* [46] and *Selenium WebDriver* [47]. They also offer benefits in allowing to programmatically define and execute user actions within the app, further streamlining the testing process and minimizing the need for manual interventions.

Our future works will also enhance the algorithms supporting the *image-based UI testing method*. We aim to strengthen its reliability across a broader spectrum of UI designs and user interactions by optimizing the algorithmic performances and expanding the method's capabilities. Additionally, we plan to introduce new hands-on *Flutter* project exercises that explore more advanced concepts of *Dart* and *Flutter* applications. These exercises will incorporate real-world scenarios, challenging students to apply their knowledge in practical settings and deepen their understanding of *Flutter* programming principles.

6.4.4. Advantages for University Teachers

The *Flutter Programming Learning Assistant System (FPLAS)* offers significant advantages for university teachers. By automating the assessments of student projects, *FPLAS* reduces the manual grading workloads, allowing teachers to focus more on interactive and engaging teaching methods. Additionally, the system provides quick and accurate feedback to students, which enhances the overall learning experience and supports effective education in lectures and exercises.

In conclusion, this research contributes to improving educational experiences for both students and teachers by providing a robust framework for automating evaluations of *Flutter* programming exercises. The positive outcomes observed in our evaluations with 44 engineering students at Okayama University validate the effectiveness of this solution in supporting *Flutter* programming education. Through ongoing refinements and expansions, we will aspire to establish effective and scalable methods in *UI* testing within educational contexts.

7. Conclusions

In this paper, we presented the implementation of an *Image-based UI testing method* using the *Flask* framework, along with *ORB* and *SIFT* algorithms from the *OpenCV* library, to automate the assessment of students' code submissions. This contribution not only provides a conducive learning environment for *Flutter* programming but also simplifies the evaluation process, thereby enhancing the educational experiences for both students and instructors. For evaluations, we applied the proposed method to assess the students' solutions, executing their answer code files for five *Flutter* exercise projects of students at Okayama University, Japan. The positive outcomes underscore our approach's effectiveness, validating our solution's practicality and efficacy in supporting *Flutter* programming education. In future works, we will enhance the *Flutter Programming Learning Assistant System (FPLAS)* functionalities and the *image-based UI testing method* to better support students and teachers. Developments will include advanced analytics features to track student progress, adaptive learning pathways, refined algorithms, and expanded capabilities of the UI testing method. We will also introduce new hands-on *Flutter* project exercises covering more advanced concepts of *Dart* and *Flutter* applications. By incorporating real-world scenarios, we aim to deepen students' understanding of *Flutter* programming. Moreover, the successful implementation and positive outcomes observed at Okayama University in Japan will provide a model that can be replicated and customized for use at other universities.

Author Contributions: Conceptualization, S.T.A. and N.F.; Methodology, S.T.A.; Software, S.T.A., L.H.A., S.A.K., K.H.W. and M.M.; Investigation, S.T.A. and L.H.A.; Writing—original draft, S.T.A.; Writing—review & editing, S.T.A. and N.F.; Visualization, S.T.A., L.H.A., S.A.K., K.H.W. and M.M.; Supervision, N.F.All authors have read and agreed to the published version of the manuscript

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding authors.

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

References

1. Criollo-C, S.; Guerrero-Arias, A.; Jaramillo-Alcázar, Á.; Luján-Mora, S. Mobile Learning Technologies for Education: Benefits and Pending Issues. *Appl. Sci.* **2021**, *11*, 4111. [CrossRef]
2. McQuiggan, S.; Kosturko, L.; McQuiggan, J.; Sabourin, J. *Mobile Learning: A Handbook for Developers, Educators, and Learners*; Wiley: Hoboken, NJ, USA, 2015.
3. Flutter. Available online: https://docs.flutter.dev/ (accessed on 1 June 2024).
4. Dart. Available online: https://dart.dev/overview/ (accessed on 1 June 2024).
5. Aung, S.T.; Funabiki, N.; Aung, L.H.; Kinari, S.A.; Mentari, M.; Wai, K.H. A Study of Learning Environment for Initiating Flutter App Development Using Docker. *Information* **2024**, *15*, 191. [CrossRef]
6. Jackson, S.; Wurst, K.R. Teaching with VS code DevContainers: Conference workshop. *J. Comput. Sci. Coll.* **2022**, *37*, 81–82.
7. Khan, S.; Usman, R.; Haider, W.; Haider, S.M.; Lal, A.; Kohari, A.Q. E-Education Application using Flutter: Concepts and Methods. In Proceedings of the 2023 Global Conference on Wireless and Optical Technologies (GCWOT), Malaga, Spain, 24–27 January 2023; pp. 1–10. [CrossRef]
8. Boada, I.; Soler, J.; Prados, F.; Poch, J. A teaching/learning support tool for introductory programming courses. In Proceedings of the Information Technology Based Proceedings of the Fifth International Conference on Higher Education and Training (ITHET), Istanbul, Turkey, 31 May–2 June 2004; pp. 604–609. [CrossRef]
9. Crow, T.; Luxton-Reilly, A.; Wuensche, B. Intelligent Tutoring Systems for Programming Education: A Systematic Review. In Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Australia, 30 January–2 February 2018; pp. 53–62. [CrossRef]
10. Keuning, H.; Jeuring, J.; Heeren, B. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* **2018**, *19*, 1–43. [CrossRef]

11. Sun, X.; Li, T.; Xu, J. UI Components Recognition System Based On Image Understanding. In Proceedings of the 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 11–14 December 2020; pp. 65–71. [CrossRef]
12. CNN. Available online: https://en.wikipedia.org/wiki/Convolutional_neural_network (accessed on 1 June 2024).
13. Khaliq, Z.; Farooq, S.U.; Khan, D.A. A Deep Learning-based Automated Framework for Functional User Interface Testing. *Inf. Softw. Technol.* **2022**, *150*, 13. [CrossRef]
14. Wang, W.; Lam, W.; Xie, T. An infrastructure approach to improving effectiveness of Android UI testing tools. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual, Denmark, 11–17 July 2021; pp. 165–176. [CrossRef]
15. UIAutomator. Available online: https://developer.android.com/training/testing/other-components/ui-automator (accessed on 1 June 2024).
16. Tareen, S.A.K.; Saleem, Z. A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. In Proceedings of the 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), Sukkur, Pakistan, 3–4 March 2018; pp. 1–10. [CrossRef]
17. Zhong, B.; Li, Y. Image Feature Point Matching Based on Improved SIFT Algorithm. In Proceedings of the 2019 IEEE 4th International Conference on Image, Vision and Computing (ICIVC), Xiamen, China, 5–7 July 2019; pp. 489–493. [CrossRef]
18. Gupta, S.; Kumar, M.; Garg, A. Improved object recognition results using SIFT and ORB feature detector. *Multimed. Tool. Appl.* **2019**, *78*, 34157–34171. [CrossRef]
19. Andrianova, E.G.; Demidova, L.A. An Approach to Image Matching Based on SIFT and ORB Algorithms. In Proceedings of the 2021 3rd International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA), Lipetsk, Russian Federation, 10–12 November 2021; pp. 534–539. [CrossRef]
20. Chhabra, P.; Garg, N.K.; Kumar, M. Content-based image retrieval system using ORB and SIFT features. *Neur. Comput. Applic.* **2020**, *32*, 2725–2733. [CrossRef]
21. Flask. Available online: https://flask.palletsprojects.com/en/3.0.x/ (accessed on 1 June 2024).
22. OpenCV. Available online: https://docs.opencv.org/4.x/ (accessed on 1 June 2024).
23. ORB. Available online: https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html (accessed on 1 June 2024).
24. SIFT. Available online: https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html (accessed on 1 June 2024).
25. GitHub. Available online: https://docs.github.com/en (accessed on 1 June 2024).
26. Moodle. Available online: https://moodle.org/ (accessed on 1 June 2024).
27. Docker. Available online: https://docs.docker.com/get-started/overview/ (accessed on 1 June 2024).
28. Visual Studio Code. Available online: https://code.visualstudio.com/docs (accessed on 1 June 2024).
29. OS. Available online: https://docs.python.org/3/library/os.html (accessed on 1 June 2024).
30. Shutil. Available online: https://docs.python.org/3/library/shutil.html (accessed on 1 June 2024).
31. Subprocess. Available online: https://docs.python.org/3/library/subprocess.html (accessed on 1 June 2024).
32. Xdotool. Available online: https://pypi.org/project/xdotool/ (accessed on 1 June 2024).
33. PyAutoGUI. Available online: https://pypi.org/project/PyAutoGUI/ (accessed on 1 June 2024).
34. PIL. Available online: https://pillow.readthedocs.io/en/stable/ (accessed on 1 June 2024).
35. Numpy. Available online: https://numpy.org/ (accessed on 1 June 2024).
36. Pytesseract. Available online: https://pypi.org/project/pytesseract/ (accessed on 1 June 2024).
37. FLANN. Available online: https://docs.opencv.org/3.4/d5/d6f/tutorial_feature_flann_matcher.html (accessed on 1 June 2024).
38. BFMatcher. Available online: https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html (accessed on 1 June 2024).
39. Muuli, E.; Tonisson, E.; Lepp, M.; Luik, P.; Palts, T.; Suviste, R.; Papli, K.; Sade, M. Using Image Recognition to Automatically Assess Programming Tasks with Graphical Output. *Educ. Inf. Technol.* **2020**, *25*, 5185–5203. [CrossRef]
40. Combefis, S. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* **2022**, *1*, 3–30. [CrossRef]
41. Mozgovoy, M.; Pyshkin, E. Unity Application Testing Automation with Appium and Image Recognition. *Commun. Comput. Inf. Sci.* **2018**, *779*, 139–150. [CrossRef]
42. Rainforest QA. Available online: https://www.rainforestqa.com/blog/ui-testing-tools (accessed on 20 July 2024).
43. Applitools Eyes. Available online: https://applitools.com/platform/eyes/ (accessed on 20 July 2024).
44. Screenster. Available online: https://www.screenster.io/ui-testing-automation-tools-and-frameworks/ (accessed on 20 July 2024).
45. Flutter Drive. Available online: https://fig.io/manual/flutter/drive (accessed on 1 June 2024).
46. Appium. Available online: https://appium.io/docs/en/latest/ (accessed on 1 June 2024).
47. Selenium WebDriver. Available online: https://www.selenium.dev/documentation/webdriver/ (accessed on 1 June 2024).