2024 IEEE 4th International Conference on Digital Twins and Parallel Intelligence (DTPI) | 979-8-3503-4925-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/DTPI61353.2024.10778901

Test-Agent: A Multimodal App Automation Testing Framework Based on the Large Language Model

Youwei Li
Institute of Social Computing
Academy of Cyber
Beijing, China
liyouwei@1010club.cn

Yangyang Li*
Institute of Social Computing
Academy of Cyber
Beijing, China
liyangyang@ict.ac.cn

Yangzhao Yang
Institute of Social Computing
Academy of Cyber
Beijing, China
forester@mail.ustc.edu.cn

***Abstract*—This paper introduces a multimodal agent-based app automation testing framework, named Test-Agent, built on the Large Language Model (LLM), designed to address the growing challenges in mobile application automation testing. As mobile applications become more prevalent and emerging systems like Harmony OS Next and mini-programs emerge, traditional automated testing methods, which depend on manually crafting test cases and scripts, are no longer sufficient for cross-platform compatibility and complex interaction logic. The Test-Agent framework employs artificial intelligence technologies to analyze application interface screenshots and user natural language instructions. Combined with deep learning models, it automatically generates and executes test actions on mobile devices. This innovative approach eliminates the need for pre-written test scripts or backend system access, relying solely on screenshots and UI structure information. It achieves cross-platform and cross-application universality, significantly reducing the workload of test case writing, enhancing test execution efficiency, and strengthening cross-platform adaptability. Test-Agent offers an innovative and efficient solution for automated testing of mobile applications.**

***Index Terms*—App Automation Testing, Large Language Model, Agent**

I. Introduction

With the popularity and rapid development of mobile applications, automated App testing has become one of the key technologies to ensure application quality and improve development efficiency. However, with the emergence of new mobile systems (such as HarmonyOS Next) and new forms of application such as applets, automated testing faces new challenges.These challenges are not only cross-platform compatibility issues, but also complex interaction logic, state management, and requirements for application security and privacy protection [1]. In recent years, with the diversification of mobile devices, operating systems and development frameworks, the complexity of automated testing technologies has continued to increase, and the limitations of existing methods have become more and more significant [2], [3]. Traditional automated app testing methods rely on a large number of manually written test cases and scripts, requiring complex adaptation and maintenance between different devices and operating systems [4]. This approach is difficult to respond effectively to demand in the face of increasingly complex application ecosystems. To this end, many researchers have proposed model-based testing methods that automatically generate test cases by building an abstract model of the application. For example, the MobiGUITAR framework, which has received much attention in recent years, can explore generative models and automatically generate test cases through the GUI [5].In addition, with the continuous development of artificial intelligence technology, AI-driven test generation tools have gradually become a hot research direction. Such tools automatically generate test cases by learning application behavior patterns, such as automated testing based on reinforcement learning [6], [7], and test generation methods combining machine learning and deep learning [8]. Although AI technology shows great potential in the field of automated testing, current methods still have certain limitations. Many AI-driven testing tools rely on predefined models and rules, and therefore underperform when dealing with cross-platform compatibility, complex interaction scenarios, and edge situations [9]. In recent years, some researchers have tried to solve these problems by introducing more intelligent context understanding and adaptive test generation strategies [10], but still face challenges when dealing with emerging application forms such as small programs [11]. To solve these problems, we propose a multi-modal agent App automated testing framework based on LLM (Large Language Model) - Test-Agent. This framework combines the large language model and artificial intelligence technology [12] to realize intelligent understanding of applications and automated testing. Unlike traditional methods, Test-Agent does not need to write test scripts in advance, but by analyzing the screenshot of the application interface and the user's natural language instructions, using deep learning models to generate corresponding test behaviors, and automatically execute them on mobile devices. This approach does not require access to the back end of the system, and can be tested solely on screen shots and UI structure information, demonstrating high cross-platform and cross-application versatility. Test-Agent framework shows significant innovation and advantages in the following aspects:

- **No need to pre-write test cases:** · Traditional automated testing usually requires test engineers to write a large number of test scripts and use cases to cover a variety of

* Corresponding author.

user scenarios [13]. Test-Agent framework makes use of the powerful semantic understanding ability of LLM, and can directly analyze the test objectives and operation steps from the user's natural language instructions, avoiding the tedious test case writing and maintenance work, and greatly improving the test efficiency.

- **Cross platform and cross-application universality:** The Test-Agent framework does not rely on any system backend and is based only on mobile device screenshots and UI structure information. This design makes the framework extremely versatile and can run seamlessly on different operating systems and various mobile applications, reducing complex adaptation efforts [14].
- **Reduce professional skill requirements:** · Traditional automated testing usually requires test engineers with programming ability and test scripting skills [15]. Test-agent framework only needs users to input natural language Test instructions to complete automated testing of mobile applications, which greatly reduces the requirements for professional skills.
- **Automated Testing for Mini Programs:** · As an emerging form of mobile application, small programs are nested inside host applications such as wechat, so they have the limitation of the operation environment, and the compatibility and permission issues of the client need to be considered. At the same time, the page structure and interaction logic of small programs are complex, and the element positioning is difficult. Traditional automated testing methods are not sufficient to support them. Test-Agent framework based on screen capture and UI structure information, through LLM multi-modal understanding ability, can accurately understand the interaction logic of small programs, generate appropriate Test behavior, providing a new solution for the automated testing of small programs.

II. The main method for automated testing of current apps

A. Based on Appium framework

Appium is an open-source automation testing tool for mobile applications on Android and iOS. It supports native, hybrid, and mobile web apps without requiring changes to the app's code. Appium is compatible with multiple programming languages, making it accessible and versatile.Supports multiple programming languages for writing test scripts, offering control and customizationBut setting up Appium for the first time can be challenging, especially for those unfamiliar with the environment or mobile automation in general. Configuring the necessary drivers, managing dependencies, and setting up the testing environment can require a significant amount of time and expertise [16].

B. Based on Espresso framework

Espresso is a powerful and reliable testing framework for Android applications, offering tight integration with the Android ecosystem, fast execution, and a user-friendly API. Since Espresso is specifically designed for Android, so it cannot be used for cross-platform testing. This limitation means separate frameworks are needed for iOS or other platforms.While basic tests are easy to write, more complex scenarios may require a deeper understanding of Espresso's API and Android internals, which can be challenging for beginners. [17].

C. Based on XCUITEST framework

XCUITest is a powerful and efficient UI testing framework for iOS applications, offering deep integration with Xcode, high reliability, and fast execution. But XCUITest is limited to iOS and requires Xcode to run, which restricts its use to macOS environments and makes it unsuitable for cross-platform testing. Additionally, as a white-box testing framework, it requires access to the app's source code, which may not be feasible for external testing scenarios. While basic tests are straightforward, more complex interactions can have a steep learning curve, particularly for those new to iOS testing [18].

D. Robot Framework + Appium

The integration of Robot Framework with Appium provides a powerful solution for automating mobile application testing across Android and iOS platforms [19]. This combination leverages the readability and maintainability of keyword-driven tests with the cross-platform capabilities of Appium. The combination of Robot Framework and Appium can introduce complexity in setup and configuration, especially for beginners. The initial learning curve for understanding how to effectively use keywords, integrate with Appium, and manage dependencies can be steep.some advanced mobile-specific interactions may require custom keywords or additional scripting [20].

E. GUI Test

Vision-based mobile app GUI testing offers a promising advancement by enhancing the robustness and flexibility of traditional GUI testing methods [21]. However, it does not eliminate the need for writing and maintaining test scripts, which can still be complex and resource-intensive.it still requires ongoing script maintenance and updates, particularly as the app evolves. It's evident that existing mobile app automation testing methods have several limitations. These include the need for extensive script maintenance, challenges with cross-platform compatibility, and the complexity of setup and configuration. These issues highlight the necessity for a more intelligent, adaptable, and universal automation testing solution—one that can seamlessly handle diverse UI changes, reduce the need for manual intervention, and offer robust performance across different platforms and devices. Such a solution would significantly improve the efficiency and effectiveness of mobile app testing, meeting the growing demands of modern software development. .

III. Framework design and implementation

The Test-Agent framework is divided into five modules.

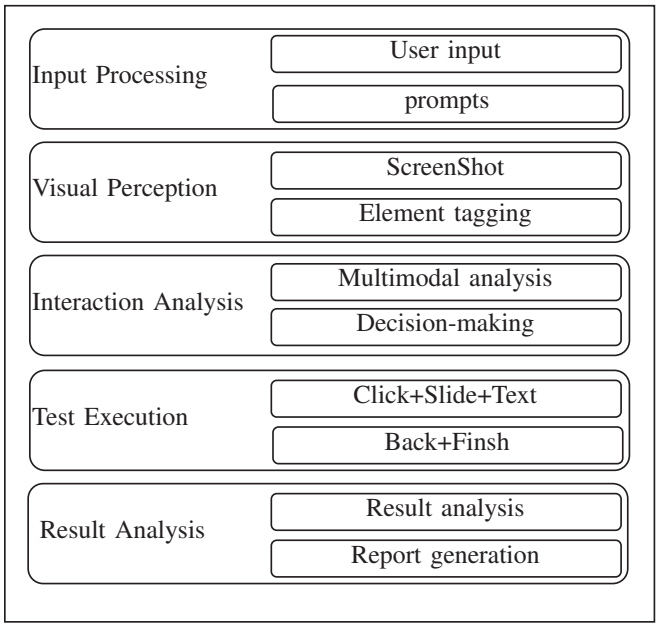

Fig. 1. Example of the overall structural diagram.

A. Input processing module

The module receives user input natural language instructions through a natural language parser, and uses the sentence vector generation technology of these advanced models to deeply understand the overall semantics of the instructions. Then, through named entity recognition (NER) technology [22], the model can identify key entities in the instruction, such as function names, data fields, operation objects, etc., which are the basic elements constituting the test steps. The model then uses its learning of large amounts of text data, combined with intent classification algorithms, to analyze and understand the user's true intent. Based on these understood and identified key entities, the model translates natural language instructions into a series of continuous, logically clear test steps. These steps might include opening a screen, entering specific data, performing an action, verifying that the results are as expected, and so on.

B. Visual perception module

This module is responsible for extracting the screenshot and UI structure information from the mobile device, extracting the elements to sort and tag, and then uploading the screenshot to the LLM service. · Visual element recognition: This component utilizes computer vision technologies such as object detection and image segmentation to analyze screenshots and automatically recognize various interactive elements on the interface, such as buttons, input boxes, text, etc. · ScreenShot: Capture screenshots using adb command. · Pretreatment: Adjust the image size, convert to grayscale, and use Gaussian filtering to remove noise,Weighted average method can be used. $Gray=0.299\times Red+0.587\times Green+0.114\times Blue$ · Edge detection: Using edge detection algorithms to identify edges in images. · Color segmentation: Segmenting an image into different regions based on color differences. · Element extraction: Extract UI elements by combining edge detection and color segmentation results. · Element Collection: Store the extracted elements in a data dictionary. · Traverse Sorting: Traverse the collected elements and sort them according to specific positions · Element tagging: Assign unique identifiers to sorted elements. Finally, upload the marked images to the interaction analysis module.

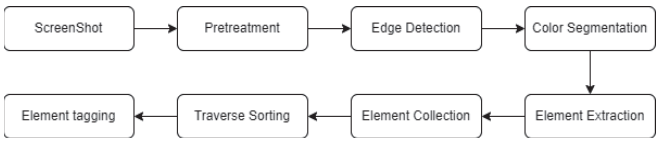

Fig. 2. Example of the overall structural diagram.

C. Interaction analysis module

The interactive analysis module is the core part of the framework. Based on the large language model technology with multi-modal capability, it analyzes the application interface screenshot and the natural language instructions input by users, identifies and analyzes the interface elements and their interaction logic, and finally outputs the execution action instructions. Specifically, it includes three important parts [23], [24]:

- Visual element recognition: This component utilizes computer vision technologies such as object detection and image segmentation to analyze screenshots and automatically recognize various interactive elements on the interface, such as buttons, input boxes, text, etc.
- Semantic understanding: This component utilizes natural language processing technology to perform semantic analysis on user input instructions and extract key interaction objects, actions, and other information. By combining the outputs of these two components, we can construct an interaction model for the application interface, including interface elements and their attributes, as well as the logical relationships between elements. This provides important input for subsequent test behavior generation.
- Instruction output: This component utilizes Prompt prompts to output corresponding action instructions and generate corresponding test steps, such as clicking, sliding, and other operations.

D. Test execution module

This model utilizes deep reinforcement learning technology to automatically generate corresponding test behavior sequences. The block receives the test steps generated by the user interaction analysis module and automatically performs corresponding operations on the mobile device, such as clicking, sliding, etc., through the ADB tool.

E. Result analysis module

This module monitors the execution process, collects and analyzes the results of each step, discovers and records problems and exceptions that occur during the testing process, takes screenshots of the exception interface, and generates detailed test reports.

The entire testing process is as follows:

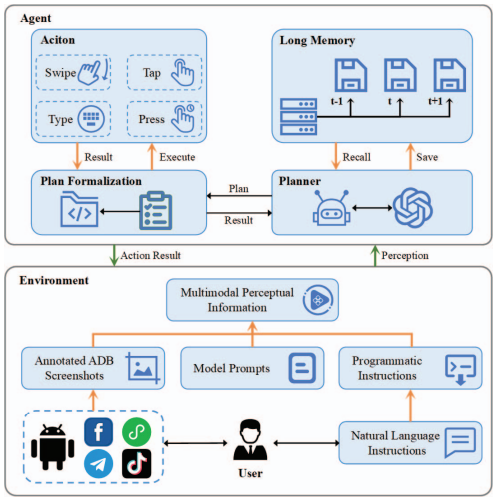

Fig. 3. Entire test flow chart.

Users input natural language directives, such as ”testing the ordering function of a certain external app.” Input processing module: Natural language conversion directives, decomposed into response steps:

- Start the application.
- Browsing and selecting merchants.
- Browsing and selecting delicious food.
- Click to purchase.

This process is represented by the following formula:

$$F = \{S, A, T, R\}$$

-*S* represents the current state, including screenshots and UI structure information

-*A* represents executable test actions, such as clicking, sliding, etc

-*T* represents a state transition function that maps the current state and test action to the next state

-*R* represents a reward function used to evaluate the effectiveness of test actions

The goal of this framework is to automatically generate the optimal test sequence by maximizing the reward function *R*:

$$Max \quad a_1, a_2, ..., a_n R(s_0, a_1, s_1, a_2, ..., s_n)$$

Where s_0 represents the initial state, a_i represents the i_{th} test action, and s_i represents the i_{th} state.

To achieve this goal, we adopted a method based on the Large Language Model (LLM) to convert natural language instructions into corresponding test action sequences, which can be represented as:

$$p(a_i|s_i, instruction) \quad \infty \quad exp(LLM(s_i, instruction, a_i))$$

Among them, instruction represents the natural language instruction given by the user, and

$$(p(a_i|s_i, instruction))$$

represents the probability of executing test action a_i in the current state s_i and instruction. By iterative executing test actions and adjusting the LLM model based on feedback, an efficient automated testing solution can ultimately be obtained.

The key advantage of this framework is that it utilizes artificial intelligence technology to achieve intelligent understanding and automatic testing of mobile applications, without the need to write a large number of test scripts in advance, greatly reducing the workload of test case writing. At the same time, it does not rely on underlying system APIs, only requires screenshots and natural language instructions, and has good cross-platform adaptability.

We model the testing process as a Markov Decision Process (MDP), with the current interface screenshot as the state and various testing actions (click, slide, etc.) as the actions, to maximize testing effectiveness.

Specifically, we define the following MDP model:

$$States = \{screenshot, interactionmodel\}$$

$$Actiona = \{Click, swipe, input, ...\}$$

$$Reward \quad function \quad R(s, a) = f(coverage, errordetectionrate, ...)$$

Then, using reinforcement learning algorithms such as DQN, an intelligent agent Test-Agent is trained, which can select the optimal test action based on the current state s and user instructions, and ultimately generate a complete sequence of test behaviors.

During the training process, we also introduced heuristic Prompt prompts to guide the Test-Agent in learning more reasonable and efficient testing strategies. For example, ”Based on user instructions, interface elements should be located first, followed by clicking, inputting, and other operations, and finally checking whether the results meet expectations.” By continuously iterating and optimizing, Test-Agent can learn the best testing plan for different application scenarios.

IV. Experimental Evaluation

To verify the effectiveness of the framework, we conducted detailed testing and evaluation on multiple real mobile applications.

A. Experimental setup

Application and Scenario Selection:

The experimental equipment was selected from different manufacturers, systems, versions, and processor architectures

to cover the current mainstream usage scenarios as much as possible.Specific scenarios include:

A. Experimental setup

TABLE I
EXPERIMENTAL EQUIPMENT PARAMETERS.

Device Name	System	Version	Processor
google pixel8	Android	12	ARM
iPhone 12	IOS	14	ARM
xiaomi 10	Android	11	ARM
huawei mate60	HarmonyOS	4.0	ARM
Samsung galaxy 10	Android	12	ARM
Xiaoyao Simulator	Android	12	X86
Genymotion	Android	10	X86

The applications chosen include:

Social Media App**:** Testing common functionalities such as user login and message sending.

E-commerce App**:** Testing actions like searching for products, adding to cart, and making payments.

Utility App**:** Testing button responses and settings page modifications.

These applications are highly interactive, covering a wide range of common app functionalities to provide a well-rounded assessment of the testing frameworks.

The test scenarios include:

- User login/logout
- Page navigation and element clicking
- Data input and validation (e.g., text fields, buttons)
- Data processing and display (e.g., lists, charts).

These scenarios simulate real user interactions, ensuring the experiment reflects typical app testing conditions.

Evaluation Metrics:

To objectively evaluate the performance of each framework, the following criteria were used:

Test case creation time: The time required to write and complete the test cases (measured in hours or minutes).

Lines of code (LOC): The number of lines needed to implement the full test case.

Code readability: Assessed by external experts, focusing on simplicity and maintainability.

Framework adaptability: The ability of each framework to support both iOS and Android platforms.

These metrics collectively measure developer workload, framework usability, and cross-platform adaptability, providing a comprehensive evaluation.

Test Case Development Process:

For each application and scenario, the same test cases were designed and implemented in each framework. The development process was conducted under identical conditions, including the same development environments and hardware, ensuring fairness. The developers involved in the process had the same level of experience to guarantee that each framework was used optimally. The development process followed these steps:

Test requirements definition: Identifying core functionality to be tested for each app.

Writing the test cases: Implementing the test cases using each framework.

Test execution: Running and recording the performance of the test cases.

Interaction Methods:

During test case development, the developers used the standard development tools for each framework (e.g., Android Studio, Xcode), adhering to the official documentation and best practices. When encountering issues, developers were allowed to refer to official resources or collaborate within the team, but no external assistance was permitted, simulating a real-world development environment.

B. Experimental result

The results in terms of test case creation time, lines of code, and other criteria are summarized in the table below:

TABLE II
WRITING TEST CASES TAKES TIME (HOURS)

Framework	Test Case Creation Time	Lines of Code	Adaptability
Appium	3	265	Android/iOS
Espresso	3.2	290	Android-only
XcuiTest	3.0	220	iOS-only
Robot Framework	2.9	205	Platform-dependent
GUI	2.0	173	Platform-dependent
Test-Agent	0.1	2	All platforms

1) Test case writing workload: From the data, it is evident that the Test-Agent significantly outperformed the existing mainstream frameworks in terms of both time and lines of code. The new framework completed the test cases in just 10 minutes with 50 lines of code, whereas other frameworks required several hours and hundreds of lines of code. Appium, though cross-platform, took the longest time and required the most lines of code due to its comprehensive API, while Espresso and XCUITest were restricted to their respective platforms (Android and iOS), limiting their flexibility. In terms of code readability, external experts evaluated Test-Agent code as more concise and intuitive, making it easier to maintain. Appium's complexity, on the other hand, resulted in lower readability due to its extensive APIs and detailed setup. Regarding adaptability, Test-Agent supported both Android and iOS platforms, making it a versatile solution, whereas Espresso and XCUITest were limited to specific platforms, reducing their usability in cross-platform projects.

TABLE III
WRITING TEST CASES CODE LINES

	login function	purchase product	post function
Appium	92%	91%	93%
Espresso	93%	92%	93%
XcuiTest	92%	93%	92%
Robot Framework	92%	87%	93%
GUI	91%	89%	92%
Test-Agent	97%	97%	96%

2) Test task success rate: There are certain cases of testing process failures using current mainstream methods, and the success rate of Test-Agent testing tasks is significantly higher

than other app automation testing methods, with very few failures.

Based on the experiment, we reached the following conclusions:

Dramatic improvement in development efficiency: Test-Agent eliminates the need for writing test cases, significantly reducing development time and code lines compared to mainstream frameworks, particularly appealing for projects requiring rapid iteration. Outstanding cross-platform compatibility: Test-Agent supports all major mobile systems including Android and iOS, streamlining efforts and enhancing development efficiency, in contrast to limited cross-platform capabilities of frameworks like Espresso and XCUITest. Lowered professional skill requirements: The straightforward code structure simplifies the use and maintenance of Test-Agent, significantly lowering the skill set needed for its operation.

V. Conclusion and future work

This article proposes a multimodal agent automation testing framework called Test-Agent based on the Large Language Model (LLM), which utilizes artificial intelligence technology to achieve intelligent understanding and automatic testing of applications. This framework does not require a large number of pre-written test scripts, but instead generates corresponding test behaviors using deep learning models by analyzing application interface screenshots and user natural language instructions, and automatically executes them on mobile devices.

In the future, we will further optimize the framework, improve its robustness and intelligence level, enhance its ability to handle complex interaction logic and state management and explore its application in a wider range of scenarios, providing a new solution for the field of mobile application automation testing.

References

[1] Santos I, C Filho J C, Souza S R S. A survey on the practices of mobile application testing[C]//2020 XLVI Latin American Computing Conference (CLEI). IEEE, 2020: 232-241.

[2] Heitkötter H, Hanschke S, Majchrzak T A. Evaluating cross-platform development approaches for mobile applications[C]//Web Information Systems and Technologies: 8th International Conference, WEBIST 2012, Porto, Portugal, April 18-21, 2012, Revised Selected Papers 8. Springer Berlin Heidelberg, 2013: 120-138.

[3] Zohud T, Zein S. Cross-platform mobile app development in industry: A multiple case-study[J]. International Journal of Computing, 2021, 20(1): 46-54.

[4] Kong P, Li L, Gao J, et al. Automated testing of android apps: A systematic literature review[J]. IEEE Transactions on Reliability, 2018, 68(1): 45-66.

[5] Yu S, Fang C, Tuo Z, et al. Vision-Based Mobile App GUI Testing: A Survey[J]. arXiv preprint arXiv:2310.13518, 2023.

[6] Adamo D, Khan M K, Koppula S, et al. Reinforcement learning for android gui testing[C]//Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. 2018: 2-8.

[7] Koroglu Y, Sen A, Muslu O, et al. Qbe: Qlearning-based exploration of android applications[C]//2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018: 105-115.

[8] Li Y, Yang Z, Guo Y, et al. A Deep Learning Based Approach to Automated Android App Testing. arXiv 2019[J]. arXiv preprint arXiv:1901.02633.

[9] Khaliq Z, Farooq S U, Khan D A. Artificial intelligence in software testing: Impact, problems, challenges and prospect[J]. arXiv preprint arXiv:2201.05371, 2022.

[10] Liu Z, Chen C, Wang J, et al. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-12.

[11] Wang T, Xu Q, Chang X, et al. Characterizing and detecting bugs in wechat mini-programs[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 363-375.

[12] Carolan K, Fennelly L, Smeaton A F. A Review of Multi-Modal Large Language and Vision Models[J]. arXiv preprint arXiv:2404.01322, 2024.

[13] Yu S, Fang C, Li T, et al. Automated mobile app test script intent generation via image and code understanding[J]. arXiv preprint arXiv:2107.05165, 2021.

[14] Berihun N G, Dongmo C, Van der Poll J A. The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review[J]. Computers, 2023, 12(5): 97.

[15] Umar M A, Zhanfang C. A study of automated software testing: Automation tools and frameworks[J]. International Journal of Computer Science Engineering (IJCSE), 2019, 6(217-225): 47-48.

[16] Singh S, Gadgil R, Chudgor A. Automated testing of mobile applications using scripting technique: A study on appium[J]. International Journal of Current Engineering and Technology (IJCET), 2014, 4(5): 3627-3630.

[17] Ardito L, Coppola R, Morisio M, et al. Espresso vs. eyeautomate: An experiment for the comparison of two generations of android gui testing[C]//Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering. 2019: 13-22..

[18] El-Morabea K, El-Garem H, El-Morabea K, et al. Testing pyramid[J]. Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS, 2021: 65-83.

[19] Cruz L, Abreu R. On the energy footprint of mobile testing frameworks[J]. IEEE Transactions on Software Engineering, 2019, 47(10): 2260-2271.

[20] Thejashwini S, Kumar M S, Alex S A. Ims based session initiation protocol in robot framework for telephony services[C]//2018 International Conference on Inventive Research in Computing Applications (ICIRCA). IEEE, 2018: 1218-1223.

[21] Yu S, Fang C, Tuo Z, et al. Vision-Based Mobile App GUI Testing: A Survey[J]. arXiv preprint arXiv:2310.13518, 2023.

[22] Song, Natural Language Processing (Almost) from Scratch" by Ronan Collobert, Jason Weston,L'eon Bottou, Michael Karlen, Koray Kavukcuoglu, Pavel Kuksa.

[23] DemoGPT. See Github.com/melih-unsal/Demo website, 2023.

[24] GPT-engineer. See Github.com/AntonOsika/gpt website, 2023.