AutoTestGPT: A System for the Automated Generation of Software Test Cases based on ChatGPT

Hui Liu[1], Liang Liu[1], Caijie Yue[1], Yang Wang[1*], and Bo Deng[2]

[1] Suzhou Technological Garden, Aerospace Information Research Institute, Chinese Academy of Sciences, Dushu Lake Road No.158, Suzhou Industrial Park, Jiangsu Province, 215000, China.

[2] Aerospace Information Research Institute, Chinese Academy of Sciences, Dengzhuang South Road No.9, Haidian District, Beijing, 100094, China.

* Corresponding author. Tel.: 0512-69836920; email: wangyang@aircas.ac.cn
Manuscript submitted July 25, 2024; revised November 20, 2024; accepted November 26, 2024; published December 20, 2024.
doi: 10.17706/jsw.19.4.108-120

Abstract: The design and generation of software test cases stand as critical steps in elevating the levels of automation and intelligence in software testing. Given the robust natural language understanding and code generation capabilities of ChatGPT, this paper, after extensive research on ChatGPT's applications in the field of software testing in recent years, introduces a ChatGPT-based software test case auto-generation system named AutoTestGPT. This system leverages ChatGPT as its intelligent engine to conduct dialogue training. It extracts key information from structured testing requirements, leading to the formulation of comprehensive testing plans. Subsequently, it systematically generates corresponding test cases according to the devised plans. Finally, the system executes the generated test cases, conducts result verification, and generates detailed testing reports. Experimental results within the API testing framework and test case generation demonstrate that the API testing framework generated using AutoTestGPT exhibits high usability. In comparison to manually coding and constructing test frameworks, the time required for test framework generation is reduced by over 70%. AutoTestGPT demonstrates high efficiency in handling complex test case generation tasks, thereby enhancing the automation and intelligence levels in test case generation. This system lays a robust foundation for the establishment of intelligent systems in software testing for the future.

Keywords: automated test case generation, ChatGPT, intelligent testing.

1. Introduction

Software testing plays a pivotal role in ensuring software quality and reliability, with the design and generation of test cases constituting its core and essence [1]. Traditional software testing often relies on manual efforts by testing personnel to write unit test cases, construct automated API testing frameworks, and perform manual Web UI testing. Manual testing is time-consuming, tedious, and repetitive. Conventional methods of test case design require significant human investment in terms of time and effort. The efficiency of test case generation is low, and the extent of test coverage often relies heavily on the experience of the test case designers. Despite the emergence of automated testing, which to some extent addresses these issues, it still necessitates manual coding of test scripts. The establishment of automated testing frameworks [2] also depends on the coding capabilities of testing personnel.

To surmount these challenges, researchers have been continuously exploring the combination of artificial intelligence algorithms and automation techniques to enhance the generation of software test cases [3, 4]. ChatGPT (Chat Generative Pre-trained Transformer) is a conversational large-scale language model [5] released by the artificial intelligence research company OpenAI. Through extensive pre-training(the model is trained on large amounts of data in advance to learn common features) and fine-tuning(fine-tune pre-trained models on small data sets for specific tasks to optimize performance) on vast text datasets, it is capable of generating coherent, grammatically sound text with a certain level of semantic understanding and robust code generation and interpretation capabilities[6, 7]. Recognizing the unique features of ChatGPT, this paper focuses on leveraging the ChatGPT model to elevate the level of software test case generation. Specifically, this paper introduces an AutoTestGPT system for automated software test case generation, utilizing the ChatGPT model as an intelligent engine. It engages in dialog training with prompts (users ask questions to large models) [8] used across different stages of the testing cycle and generates standardized conversation templates. Subsequently, it dissects testing requirements and specifications into test tasks, formulating comprehensive test plans. Based on these plans, it systematically generates test cases and finally validates the generated test cases' results and produces testing reports.

The proposed AutoTestGPT test case generation system in this paper is applicable to common testing types, such as unit testing, API testing and UI testing. This study focuses on the frequently encountered scenario of API testing and designs experiments. The AutoTestGPT system is utilized to generate automated API testing frameworks and corresponding test cases. The experimental results demonstrate that the AutoTestGPT system aligns with the requirements for test case design, including correctness and reusability of framework generation. Compared to manual framework composition, the efficiency of framework generation increases by over 70%, significantly alleviating the burden of test case and framework development for testing personnel.

2. Related Works

Benefiting from the rapid advancement of artificial intelligence technology, researchers have extensively employed search based algorithms such as simulated annealing algorithm [9], genetic algorithm [10, 11], particle swarm optimization algorithm [12], AI algorithms such as machine learning [13–15] and deep learning methods to automatically generate software test cases. This approach helps to overcome limitations of traditional test case generation methods. For instance, Jana [16] introduced a test case generation method called DeepTest based on convolutional neural networks, which was applied to test deep neural networks in autonomous driving vehicles. Guo [17] proposed an automatic test case generation framework based on Generative Adversarial Networks (GAN). This novel framework was applied for generating test cases in the unit testing and integration testing phases. However, deep learning methods have a higher entry barrier and may not fulfill the requirement for convenient and rapid test case generation. Therefore, in recent years, the highly regarded ChatGPT conversational generative model has captured the attention of researchers as an alternative approach.

ChatGPT is constructed upon the foundational architecture of the Transformer, a large-scale language model designed for dialogues, employing a pre-training and generative approach. Drawing upon ChatGPT's formidable capabilities in natural language comprehension [18], error correction [19], and contextual learning [20, 21], researchers have extended its application across diverse domains, including education, healthcare, law, academic publishing, and software engineering [22]. In the context of software testing, ChatGPT is commonly employed for tasks such as automated program repair, resolution of programming bugs, identification of failing test cases, and education within the realm of software testing. For instance, Sobania *et al.* [23] applied ChatGPT to automated program repair techniques, assessing its repair

performance on a standard error repair benchmark. Surameery [24] investigated the utilization of ChatGPT for addressing programming bugs, emphasizing its potential as an integral component of a comprehensive debugging toolkit. T.-O. [25] employed ChatGPT for automated software fault detection. Jalil [26] explored the application of ChatGPT within the domain of software testing education.

The current spectrum of ChatGPT's applications encompasses various aspects of the software testing domain [27]. In terms of software test case generation, Zimmermann [28] studied the technology of using GPT-3 to generate test cases in the field of GUI testing, and realized the function of non-technical users easily entering test cases for desktop and mobile applications. However, for many types of test in the field of software testing, it is difficult to have a general system design idea for the field of test case generation. Building upon the aforementioned studies, this article introduces the AutoTestGPT test case generation system. This system adeptly employs large language models within the domain of software test case generation, effectively addressing the painstaking process of manual test case creation and bolstering the efficiency of constructing test cases and establishing testing frameworks. Through the iterative decomposition of intricate tasks and the employment of prompt-driven dialogue training templates, the system concurrently resolves the dilemma faced by testing personnel when embarking upon prompt engineering (a complete set of interactions between the user and the large model). This advancement offers a fresh perspective on how to more extensively integrate large-scale language models such as ChatGPT into the domain of software testing, thereby expediting the automation and intellectualization of software quality assurance processes.

3. AutoTestGPT

AutoTestGPT framework mainly includes three parts: test requirements information extraction, prompt word generation and test case generation and execution. The working process is as follows. First, for a given test requirement, AutoTestGPT extracts key information of the test requirement according to different test types such as unit test, API test, UI test, etc., including test type, programming language, test framework, objective function, precision requirement and other parameters to form structured requirement information. Then input the requirement information and the predefined prompt word template into the prompt word generation section to generate the key prompt words that meet the test requirements. After the prompt words are input to the ChatGPT intelligent engine, ChatGPT systematically interprets the information related to the test task, outlines a specific test plan, and gradually designs and generates test cases according to the test plan. Finally, AutoTestGPT extracts the test case code, dynamically assembles it into executable test case scripts, iteratively executes those test cases in the appropriate test environment, and evaluates the validity of the results. When the results do not meet the requirements, the evaluation results are fed back to the prompt word generation module, a new prompt word is generated and input again to the ChatGPT intelligent engine, and the generated results are further adjusted until the results meet the test requirements. Its structure is shown in Fig. 1. The details of how this works are described in Sections 3.1, 3.2, and 3.3.

3.1. Test Requirement Information Extraction

Testing requirements often describe scenarios from a user perspective, making it challenging to precisely correspond to individual test cases. Structured requirement documents need only include three lines of content: (1) a concise description of the requirement, (2) the type of requirement, and (3) the coverage metrics specified in the requirement. Initially, these structured requirement documents are input into the AutoTestGPT system, which then extracts key information from them. The structural processing of requirement documents is completed by users before formal testing commences.

Common test types include unit testing, API testing and UI testing. Unit testing requirements mainly focus

on programming languages, test frameworks, test coverage requirements and tested functions. API testing requirements focus on interface URLs, test frameworks, test reports and additional requirements. UI testing focuses on test frameworks, page elements, positioning methods and pages. The structured information of different types of requirements is shown in Table 1, in which we define the "{{}}" slot, in which the key information extracted from the requirements needs to be filled in. Obtain the information required to generate test cases for three different test types and store them into the requirements list.

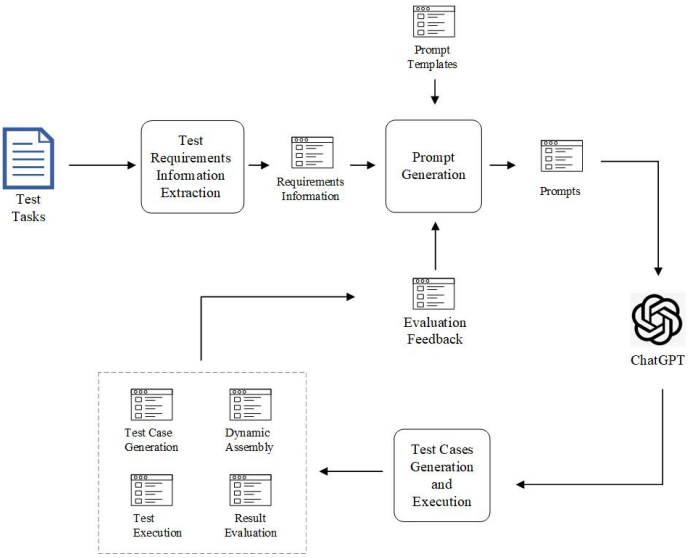

Fig. 1. Overview of AutoTestGPT.

Table 1. Test Type and Requirements Structuralization Information

Test type	Requirements information
Unit test	Given examples are as follows: a) Requirements type: {{Unit test case generation}} b) Programming language: {{}} c) Test framework: {{}} d) Test coverage: {{}} e) Test function: {{}}
API test	a) Requirements type: {{API test framework generation}} b) Interface URL: {{}} c) Test framework: {{}} d) Report type: {{}} e) Additional requirements: {{}}
UI test	a) Requirements type: {{UI test case generation}} b) Test framework: {{}} c) Page elements: {{}} d) Positioning mode: {{}} e) Operation process: {{}}

3.2. Prompt Generation

A 'Prompt' is a form of natural language input, akin to a command or instruction, designed to guide an AI model on the actions it should take or the output it should generate when performing a specific task. Throughout the entire test case generation cycle, we have devised and generated prompt examples for various stages, including test plan formulation, test case design, and test result evaluation, as illustrated in Table 2. These examples serve as the basis for training dialogues with large language models, aiding AutoTestGPT in better understanding and responding to user-provided instructions. This approach significantly enhances the model's reliability and effectiveness. The system automatically populates the extracted test requirements from Section 3.1 into the examples, thereby enabling the generation of

high-quality prompts.

3.2.1. Examples of prompts at each stage of test

Typically, a complete testing dialogue cycle consists of three stages: test plan formulation, test case generation, and test result evaluation. By analyzing the key steps in each stage, we have provided common prompt examples for each phase as shown in Table 2. The input for the test plan formulation stage is the textual description of the testing requirements, often selecting the most critical segment and placing it within the placeholder {{}}. The output encompasses test objects, test environment, test objectives, test functionality points, test methods, and test result evaluation.

In the test case generation stage, critical information extracted from Section 3.1 is utilized to generate test case script code. For unit testing, the Test framework, programming language, test function, and test coverage are successively filled into placeholders. The prompt stipulates the coverage metric for the tested function. In API testing prompts, the test framework, interface URL, parameter lists, and report type are sequentially inserted into the placeholders. To enhance the extensibility of the testing framework, the prompt additionally requests the separation of test data and test cases. In UI testing prompts, the test framework, page elements, positioning mode, and operation process are progressively entered into placeholders. Following the description of the testing process, the generation of test case code is incrementally accomplished.

The test result evaluation stage is employed to assess the usability and correctness of the test results. Three types of test results correspond to different evaluation methods and criteria, thus necessitating distinct prompts. In unit testing prompts, statistics are compiled for common coverage metrics, including statement coverage, branch coverage, and path coverage. For API testing prompts, the tested real interface is specified, and statistics are generated for interface functionality pass rate and API test case coverage. In UI testing prompts, coverage is calculated based on the proportion of executed test operations to the total number of operations.

Table 2. Examples of Prompts

Stage	Prompt Examples
Test Planning	According to the requirement {{Requirements text}}, please list the detailed test plan. The test plan is carried out from the following aspects: a) Determine the test objectives. b) Build a test environment. c) Determine the scope of the test. d) List detailed function points. e) Determine the test method. f) Define the result acceptance criteria. g) Complete the dispatch plan.
Test Case Designing	Unit test: Use {{Framework}} in {{Programming language}} to write unit test cases for the {{Function}} in the code with coverage{{Coverage}}
	API test: Use {{Framework}} to write API automation testing framework. Interface URL is {{Interface URL}} and test parameters includes {{Parameter lists}}. Test report using tools {{Report type}}. Additional requirements: {{Test data should be separated from test cases}} UI test: Use {{Framework}} to write UI Test cases for the page elements {{Page elements}}. Page elements positioning mode {{Positioning mode}}. The test operation process is as follows {{Operation process}}.
Result Evaluation	Unit test: Prompt1: Please count the statement/branch/path coverage of the unit test cases generated above.

	API test: Please verify correctness of the test framework with the actual interface. a) Actual interface URL: {{actual interface URL}}. b) Interface parameters: {{parameter lists}}. c) Expected output: {{expected output}}. d) Pass rate of result operation: {{Number of passed cases/Total number of cases}}. e) Please verify its integrity of the test framework with the actual interface. UI test: Please calculate the proportion of the passed operation steps to the total operation steps and record it as the UI test coverage.

3.2.2. Evaluation Feedback

Test case code generated by ChatGPT may exhibit certain deviations or bugs. To ensure system stability, we have designed an evaluation feedback module. This module, based on the assessment of test case results, allows AutoTestGPT to acquire execution outcomes and determine their correctness. If the results are deemed incorrect, detailed error information is retrieved and used as input for the prompt generation module. This process yields corrected test case scripts, which are then executed using the test case execution module from Section 3.3. This iterative cycle is repeated multiple times until the test cases pass completely. Prompt examples corresponding to the evaluation feedback stage are illustrated in Table 3.

Table 3. Example of Prompt during the evaluation feedback stage

Stage	Prompt Examples
Evaluation feedback	After executing the above script, the following error message is displayed: {{Error messages}}. How should I correct the test script?

3.3. Test Case Generation and Execution

With prompts tailored for each stage of the testing cycle, inputting them into ChatGPT allows for the rapid generation of corresponding output results. Following the generation of detailed test plans and test cases, AutoTestGPT automatically extracts and assembles the generated test case code into executable scripts. These scripts are then executed for the respective test cases, producing the test results.

3.3.1 Test Case Generation

For unit testing, the generated test case code, code explanations, and the method or results of unit test coverage statistics are produced. At the end of the unit test case, a method for unit test coverage statistics is generated. In API testing, the directory structure of the testing framework is initially provided. Subsequently, detailed code for each part of the API testing framework, including framework initialization, test data, reading test data configurations, actual test cases, and handling test requests and responses, is generated along with accompanying code explanations. For UI testing, a common automation testing framework, such as Selenium in Python, is invoked. Test cases for UI automation are generated based on the page source code and positioning methods.

3.3.2 Dynamic Assembly

The dynamic assembly process is a crucial step that connects the tested object with the test cases. It varies slightly for different types of testing, as outlined in Table 4. For unit testing, the dynamic assembly process begins by obtaining the source code under test and related dependencies. Subsequently, it acquires the test cases and the script for evaluating test results, ultimately completing the comprehensive assembly and generation of the test case script. In the case of API testing, following the explanation in Section 3.2, the framework for API testing is generated. Initially, the testing framework directory is established, followed by obtaining the code for each part of the testing framework. The code is then placed in script files located in different directory structures. Finally, the interface URL and parameters are obtained and filled into the testing framework configuration file, achieving the dynamic assembly of the API testing framework. For UI

testing, information about UI test cases is obtained and directly written into the test case execution file.

Table 4. Dynamic Assembly Process

Test type	Dynamic assembly process
Unit test	a) Get the source program of the function under test and related dependencies. b) Get the test cases. c) Get the test evaluation scripts. d) Assemble into test case execution file.
API test	a) Build the test framework directory. b) Create a script file in the corresponding directory, and write the code into the script files. c) Get the tested interface URL and parameters and fill them in the configuration file.
UI test	a) Get the header file information and test case code. b) Assemble into test case execution file.

3.3.3 Test Execution

Built upon commonly used means and frameworks for automated testing based on test cases, the AutoTestGPT system has preconfigured prevalent automated testing runtime environments. Once executable test case files are obtained, our system automatically executes the test cases, displaying the test case execution results on the output console. The AutoTestGPT system retrieves the execution results and further assesses whether the test scripts executed successfully. If there are no errors during execution, the system outputs the test results and a test report. In the case of execution errors, error information is obtained and used as input for Section 3.2.3, acquiring corrected test cases. This iterative loop continues until the test cases pass successfully.

3.3.4 Test Result Evaluation

For unit testing, we employ code coverage analysis methods to measure whether the test cases cover various parts of the tested code. AutoTestGPT will invoke commonly used coverage analysis tools such as the 'coverage' library in Python and the JaCoCo tool in Java to automatically generate scripts for calculating test case coverage metrics.

In the case of API testing, evaluating the correctness of the generated API testing framework involves testing with real interfaces, such as a pre-established automated API testing framework. Initially, several test cases targeting real interfaces are generated. The testing framework is then invoked to automatically execute these test cases. Finally, the pass rate of the test cases is calculated to assess the correctness of the generated API testing framework.

For UI testing, coverage metrics are designed in the prompts, primarily calculating the proportion of tested page functionalities to the total page functionalities and the proportion of correctly located elements to the total number of elements.

4. Experiment

The AutoTestGPT system is applicable to common testing types, such as unit testing, API testing, and UI testing. To validate the effectiveness of the test case generation by AutoTestGPT, we assess it using API testing as an example. As described in Section 3.2.2, evaluating the correctness of the generated API testing framework requires validation with real interfaces. To illustrate the advantages of the AutoTestGPT system in enhancing automated testing, we compared the overall time spent by five test engineers on writing and debugging an automated API testing framework with the time spent utilizing the framework generated by AutoTestGPT. This comparison serves to highlight the efficiency of the system in improving the generation of automated test cases. The ChatGPT model version chosen for this experiment is GPT-3.5.

4.1 Settings

An API testing framework serves to curtail the temporal costs associated with manual regression testing,

thereby expediting the testing cycle. The objective of this experiment is to utilize the AutoTestGPT system to invoke the widely-used API automation testing framework in Python, Pytest. The aim is to autonomously construct an API automation testing framework based on Pytest. Using a real-world interface as an exemplar, the effectiveness of the generated testing framework will be evaluated.

4.2 Procedure and Results

The experimental process for generating the API automation testing framework using the AutoTestGPT system is documented in Table 5, while the outcomes of the experiment are detailed in Table 6. Initially, the testing requirements were dissected in accordance with the provided examples, with the structured testing demands inserted into {{}} placeholders. The nature of the experiment pertains to the API Automation Test Framework, employing Python as the testing language, Pytest as the invoked testing framework, and Allure as the tool for generating testing reports. An additional stipulation entails the segregation of testing data from test cases.

4.2.1. Requirements Extraction

The original requirements for this experiment are as follows:

Please design and generate an API automation test framework based on the Python language, and call the Pytest test framework. First, establish the test framework directory structure. Secondly, the test data needs to be separated from the test cases. The test data includes the interface URL, request method, request parameters, and expected output results. Finally, test reports are required to be generated using Allure tools.

Following the testing requirement segmentation principles outlined in section 3.1, the aforementioned requirements are divided as follows:

a) Requirements type: {{API test framework generation}}

b) Interface URL: {{https://api.example.com/endpoint}}

c) Test framework: {{Pytest}}

d) Report type: {{Allure}}

e) Additional requirements: {{the test data needs to be separated from the test cases}}

4.2.2. Prompt Generation

Following the prompt generation rules and examples in Section 3.2, fill the prompt sample with the API test requirements, resulting in a prompt set for the following three stages.

Table 5. Prompt Set

Test Stage	Prompt
Test Planning	According to the requirement {{*Please design and generate an API automation test framework based on the Python language, and call the Pytest test framework.*}}, please list the detailed test plan. The test plan is carried out from the following aspects: a) Determine the test objectives. b) Build a test environment. c) Determine the scope of the test. d) List detailed function points. e) Determine the test method. f) Define the result acceptance criteria. g) Complete the dispatch plan.
Test Case Generation	Use {{Pytest}} to write API automation testing framework. Interface URL is {{https://api.example.com/endpoint}}. Test report using tools {{Allure}}. Additional requirements: {{Test data should be separated from test case}}
Result Evaluation	Please build an API automation test framework according to the above tips, and verify its correctness with the actual interface. a) Actual interface URL: {{*https://api.example.com/endpoint*}}. b) Interface parameters: {{*param1 = "value1", param2 = "value2"*}}.

c) Expected output: {{*status_code = 200, key1 = "value1", key2 = "value2"*}}.
d) Pass rate of result operation: {{*Number of passed cases/Total number of cases*}}.

4.2.3. Test Case Generation and Execution

Given the aforementioned prompt as input parameters, input into ChatGPT for the retrieval of a test plan, test framework, test case generation, and validation of test results on a real interface, the generated experimental results by AutoTestGPT are presented in Table 6. Initially, a comprehensive test plan is formulated as outlined in the Test Planning section. During the test case generation phase, AutoTestGPT proceeds as follows: First, it provides the overall design concept for the API automation test framework:

(1) Define the format of data files.

(2) Create test cases and retrieve test data from the TOML file.

(3) Implement interface requests and response handling.

(4) Generate test reports.

Subsequently, AutoTestGPT presents the directory structure of the automation testing framework, illustrated in Fig. 2. Within this structure, the test_data directory is designated for storing test data, with the file named "example_test_data.toml". In accordance with the TOML file format, ChatGPT specifies the request HOST, request method, request parameters, and expected response. The tests directory contains the test case configuration file "conftest.py", which facilitates the retrieval of test data from the TOML file, and the test case file "test_interface.py", encompassing the actual API test cases. The "api_client.py" file primarily executes the sending of interface requests and handling of responses. The "allure_report" directory is designated for storing result files from the test reports. The "pytest.ini" is the configuration file required for running pytest. Finally, the code for each section of the test framework is provided. In accordance with the output results, the construction of the API automation test framework is completed. Real API testing is conducted, and ultimately, the API test cases run successfully, validating the results.

Table 6. Results

Test Stage	Result
Test Planning	Here's the detailed test plan based on the provided requirements splitting: Test Plan ① Environment Setup Phase: • Install necessary Python packages (Pytest, Allure). • Set up the test environment on the chosen OS. ② Framework Implementation Phase: • Create the directory structure for the test framework. • Implement test cases using Pytest and separated test data. • Integrate Allure for test report generation. ③ Testing Phase: • Execute the test cases using Pytest. • Monitor and track test execution progress. ④ Reporting Phase: • Generate Allure test reports. • Review and analyze test results. ⑤ Completion Phase: • Ensure all test objectives are met. • Address any failed test cases or issues. Finalize the test framework documentation.
Test Case Generation	a) The design idea of the API automation testing framework is given. b) The project directory structure is shown in Fig. 2.
Result Evaluation	a) Build the automated test framework according to the design ideas given by AutoTestGPT. b) Correctness verification: All test cases passed the test for real interface. c) Integrity verification: All arguments contain valid equivalence classes and invalid equivalence classes.

AutoTestGPT gives the directory structure of the automated test framework:

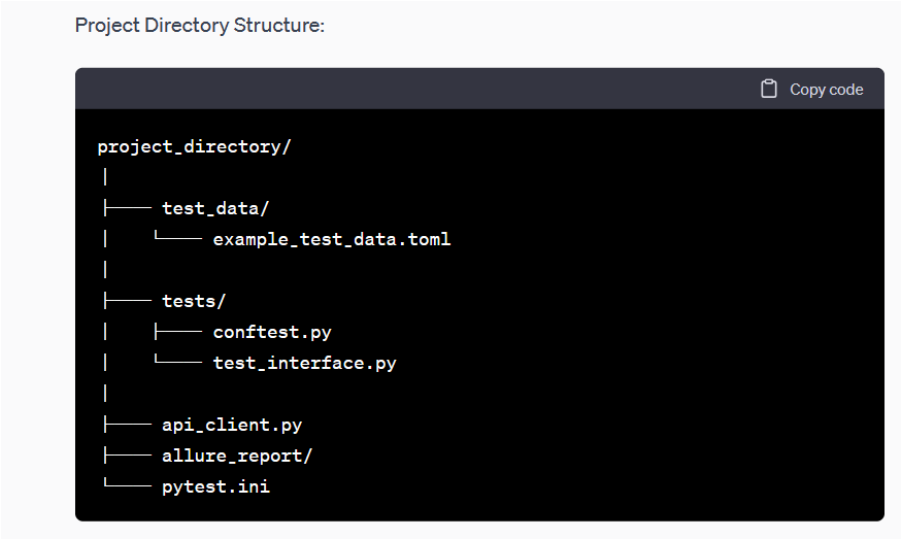

Fig. 2. Directory of the API automation testing framework generated by AutoTestGPT.

AutoTestGPT, based on the results generated in Section 4.2.2, has successfully completed the construction of the API automation testing framework. Initially, the framework directory is established according to Fig. 2, and corresponding test script files are created in the respective directories. Subsequently, the relevant code is extracted and written into the test script files, assembling them into a comprehensive API automation testing framework.

To validate the correctness of this framework, real API testing experiments are conducted by running the test framework. The experimental results indicate that the API testing framework generates a test case for each parameter, and all cases run successfully. Detection of no errors in the testing framework signifies the successful generation of a test report. The experimental outcomes affirm the practicality of the API automation testing framework generated through AutoTestGPT.

4.3 Qualitative Results

The API automation testing framework stands as a pivotal tool in enhancing the efficiency of API testing. Leveraging AutoTestGPT, the complete process of testing requirement breakdown, test plan formulation, test framework and test case generation, and validation of test results is accomplished. The resultant API automation testing framework furnishes fundamental functionalities such as automated test case composition, automated test case execution, and automated test report generation. This framework serves as the foundation upon which we need only undertake the design and implementation of test cases closely aligned with the business domain. Based on statistical data from actual projects, it takes approximately 4 hours for five experienced test development professionals to construct an API automation testing framework. In contrast, utilizing the AutoTestGPT system to build a testing framework with equivalent functionality takes about 1 hour, reducing the framework generation time by over 70%. This substantial reduction in time significantly shortens the upfront investment in the project.

5. Discussion and Analysis

The AutoTestGPT system represents a novel paradigm in the domain of software test case generation. It adeptly dissects intricate testing requisites, formulates structured prompts, fine-tunes extensive language models, and ultimately crafts test case code and frameworks of heightened utility. Its strengths encompass the following facets. 1) A salient advantage of the AutoTestGPT system resides in its amplification of test

case generation efficiency. It significantly reduces the time invested in manually composing redundant code during test case generation and framework establishment. Efficiency gains in the realm of API test framework construction have even reached an impressive 70%, signifying a substantial enhancement in testing efficacy. 2) The dialogue training module within the AutoTestGPT system addresses the challenge that testing professionals encounter when effectively formulating prompts for substantial language models. The dialogue training module furnishes comprehensive prompt templates requisite throughout the testing lifecycle, spanning from the deconstruction of testing requisites and formulation of testing plans to test case generation, evaluation of test outcomes, and report generation.

It is noteworthy that AutoTestGPT still exhibits certain limitations or areas for improvement. 1) The generation of test cases is heavily reliant on the ChatGPT intelligent engine model. The performance and stability of the ChatGPT model directly impact the performance and stability of the AutoTestGPT system. This dynamic can lead to fluctuations in the quality of code generated by the AutoTestGPT system. 2) Regarding the validation of test results, the current predominant approach involves executing the generated test cases, thereby representing a somewhat singular validation method. This deficiency underscores the need for an overarching evaluation framework to comprehensively assess outcomes, constituting a focal point for future research endeavors.

6. Conclusion

In this paper, we introduce a software test case generation system called AutoTestGPT, based on ChatGPT, to address the challenges of automated test case generation. The system extracts key information from structured test requirements, references dialogue training templates, uses ChatGPT intelligent engine to generate test plans, test cases or frameworks, then iteratively validates and automatically generates test reports. Experimental results on a typical case of API testing framework demonstrate the significant positive impact of the proposed AutoTestGPT system on improving the efficiency of test case generation and overall levels of test automation.

Future investigations shall, on one hand, be rooted in the existing AutoTestGPT system, as we delve into the assessment of the trustworthiness metrics associated with the generated test cases. On the other hand, we shall consider the use or training of a more stable iteration of the ChatGPT intelligent engine model. Such an approach serves to ensure the precision and consistency of the testing framework's output.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Hui Liu: Conceptualization, Methodology, Validation, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Project administration. Liang Liu: Conceptualization, Methodology, Validation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. Caijie Yue: Conceptualization, Methodology, Validation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. Yang Wang: Conceptualization, Methodology, Validation, Resources, Data curation, Writing – review & editing, Visualization, Supervision, Project administration. Bo Deng: Conceptualization, Methodology, Validation, Resources, Data curation, Writing – review & editing, Visualization, Supervision, Project administration; all authors had approved the final version.

Funding

This research was partially funded by the Aerospace Information Research Institute, Chinese Academy of Sciences through the "Technology system software definition and automatic evaluation technology project" grant (No. E3Z2S30200).

Acknowledgment

We would like to thank our colleagues who participated in our research, as well as the editors for their valuable feedback.

References

[1] Tung, Y. W., & Aldiwan, W. S. (2000). Automating test case generation for the new generation mission software system. In *Proc. IEEE Aerospace Conference*. Big Sky, MT, USA: IEEE.

[2] Xu, D., Xu., W., Kent, M., Thomas, L., & Wang, L., (2015). An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability*, 64(1), 247–268.

[3] Parnami, S., Sharma, K. S., & Chande, S. V. (2012). A survey on generation of test cases and test data using artificial intelligence techniques. *International Journal of Advances in Computer Networks and its Security, 2(1)*, 16–18.

[4] Khaliq, Z., Farooq, S. U., & Khan, D. A. (2022). Transformers for GUI testing: A plausible solution to automated test case generation and flaky tests. *Computer, 55(3)*, 64–73.

[5] Fraiwan, M., & Khasawneh, N. (2023). A review of ChatGPT applications in education, marketing, software engineering, and healthcare: Benefits, drawbacks, and research directions. arXiv print, arXiv: *2305.00237*. doi: 10.48550/arXiv.2305.00237

[6] Feng, Y., Vanam, S., Cherukupally, M., Zheng, W., Qiu, M., & Chen, H. (2023, June). Investigating code generation performance of ChatGPT with crowdsourcing social data. In *Proc*. *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)* (pp. 876–885). Torino, Italy: IEEE.

[7] White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2024). ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *Generative AI for Effective Software Development (pp. 71–108)*. Cham: Springer Nature Switzerland.

[8] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9), 1–35.

[9] Shirole, M., & Kumar, R. (2013). UML behavioral model based test case generation: A survey. *ACM SIGSOFT Software Engineering Notes, 38(4)*, 1–13.

[10] Haga, H., & Suehiro, A. (2012, November). Automatic test case generation based on genetic algorithm and mutation analysis. In *Proc. 2012 IEEE International Conference on Control System, Computing and Engineering* (pp. 119–123). Penang, Malaysia: IEEE.

[11] Khan, R., & Amjad, M. (2015, December). Automatic test case generation for unit software testing using genetic algorithm and mutation analysis. In *Proc. 2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)* (pp. 1–5). Allahabad, India: IEEE.

[12] Li, D., Wong, W. E., Pan, S., Koh, L. S., Li, S., & Chau, M. (2022). Automatic test case generation using many-objective search and principal component analysis. *IEEE Access*, 10, 85518–85529.

[13] Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability, 68(3)*, 1189–1212.

[14] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., & Tonella, P. (2020). Testing machine

learning based systems: a systematic mapping. *Empirical Software Engineering, 25,* 5193–5254.

[15] Tufano, M., Drain, D., Svyatkovskiy, A., & Sundaresan, N. (2022, May). Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (pp. 54–64). Pennsylvania, USA: ACM Digital Library

[16] Tian, Y., Pei, K., Jana, S., & Ray, B. (2018, May). DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 303–314). Gothenburg, Sweden: ACM

[17] Guo, X., Okamura, H., & Dohi, T. (2022). Automated software test data generation with generative adversarial networks. *IEEE Access, 10*, 20690–20700.

[18] Hariri, W. (2023). Unlocking the potential of ChatGPT: A comprehensive exploration of its applications, advantages, limitations, and future directions in natural language processing. arXiv print, arXiv:2304.02017. doi: 10.48550/arXiv.2304.02017

[19] Fang, T., Yang, S., Lan, K., Wong, D. F., Hu, J., Chao, L. S., & Zhang, Y. (2023). Is ChatGPT a highly fluent grammatical error correction system? A comprehensive evaluation. arxiv print, arxiv:2304.01746. doi: 10.48550/arXiv.2304.01746

[20] Dai, D., Sun, Y., Dong, L., Hao, Y., Ma, S., Sui, Z., & Wei, F. (2022). Why can GPT learn in-context? Language models implicitly perform gradient descent as meta-optimizers. arXiv print, arXiv:2212.10559. doi: 10.48550/arXiv.2212.10559

[21] Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., & Zhou, D. (2022). What learning algorithm is in-context learning? Investigations with linear models. arXiv print, arXiv:2211.15661. doi: 10.48550/arXiv.2211.15661

[22] Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., & Ge, B. (2023). Summary of ChatGPT-related research and perspective towards the future of large language models. *Meta-Radiology*, 100017.

[23] Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023, May). An analysis of the automatic bug fixing performance of ChatGPT. In *Proc. 2023 IEEE/ACM International Workshop on Automated Program Repair (APR)* (pp. 23–30). Melbourne, Australia: IEEE.

[24] Surameery, N. M. S., & Shakor, M. Y. (2023). Use chat GPT to solve programming bugs. *International Journal of Information Technology and Computer Engineering, (31)*, 17–22.

[25] Li, T. O., Zong, W., Wang, Y., Tian, H., Wang, Y., Cheung, S. C., & Kramer, J. (2023, September). Nuances are the key: Unlocking ChatGPT to find failure-inducing tests with differential prompting. In *Proc. 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 14–26). Luxembourg, Luxembourg: IEEE.

[26] Jalil, S., Rafi, S., LaToza, T. D., Moran, K., & Lam, W. (2023, April). ChatGPT and software testing education: Promises & perils. In *Proc. 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 4130–4137). IEEE.

[27] Bajaj, D., Goel, A., Gupta, S. C., & Batra, H. (2022). MUCE: A multilingual use case model extractor using GPT-3. *International Journal of Information Technology, 14(3)*, 1543–1554.

[28] Zimmermann, D., & Koziolek, A. (2023, April). Automating GUI-based software testing with GPT-3. In *Proc. 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 62–65). IEEE.