Approach to end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure

Evgeniy Geraskin
Institute of Computer Science and Technology
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
GeraskinEvgeniy@mail.ru

Nikita Voinov
Institute of Computer Science and Technology
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
voinov@ics2.ecd.spbstu.ru

***Abstract*—The article is devoted to end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure. The main idea is to develop a software framework to create and perform end-to-end tests written in Python.**

***Keywords*—ent-to-end testing, enterprise virtual infrastructure, configuration, software framework, Python**

I. Introduction

End-to-end testing [1] is a software testing technique that involves evaluating a system from end to end, covering every component and process, from the user interface to the database. The goal of end-to-end testing is to verify the system's functionality, performance, reliability, and security, among other aspects. End-to-end testing ensures that all parts of the system work together as expected and that there are no inconsistencies or errors that might affect the system's performance.

One of the main advantages of end-to-end testing is that it can identify potential issues that might not be detected by other testing techniques, such as unit testing or integration testing. For example, if there is a problem with the interaction between different components of the system, it may not be evident in isolation, but it could have significant consequences for the system's performance as a whole. End-to-end testing can identify such issues and help developers address them before the system is deployed.

Another benefit of end-to-end testing is that it can help reduce the time and effort required to identify and rectify errors. By testing the system as a whole, issues can be located and resolved earlier in the development process, before they become more challenging and expensive to fix. This, in turn, can help reduce the overall cost of development and improve the time-to-market for the product [2].

Moreover, conducting end-to-end testing can contribute to enhancing the product's quality and customer satisfaction. By verifying that the system functions as intended and performs well, end-to-end testing can help ensure that users have a positive experience when using the product. This improves customer loyalty and generates positive feedback.

End-to-end testing is a critical software testing technique that is essential for verifying the functionality, performance, reliability, and security of a system. By testing the system from end to end, potential issues can be identified and addressed early in the development process, leading to a higher-quality product that performs well and satisfies customers' needs. That is why it is necessary to use the end-to-end testing paradigm as an axiom, and build the testing process around it.

Similar to the software development process, end-to-end testing also follows a specific methodology. In this case, methodology refers to the principles, ideas, methods, and concepts that engineers employ while working on a project. There are currently several diverse approaches to end-to-end testing, each with its own starting points, duration of execution, and methods used at each stage. Choosing the right approach can be a challenging task, and it requires an understanding of the unique features and requirements of the system being tested.

This article focuses on the approach to end-to-end testing of a specialized software used for managing the configuration of enterprise virtual infrastructure. Later in text «application» is used to define this software. Detailed architecture of this application is described in section II.

The approach involves a comprehensive evaluation of the system from the user interface to the database. The testing process is performed in a continuous integration environment, which enables the team to test the system continually as new code is added. The testing process also includes the use of automated tests written in Python. The automated tests allow for faster and more reliable testing and enable the team to test the system across multiple platforms and configurations.

The approach also includes the use of virtual environments to simulate the production environment. This enables the team to identify potential issues that may arise in the production environment and to test the system's performance under various conditions.

II. Features of the Application Under Test and End-to-End Testing

Application for managing the configuration of enterprise virtual infrastructure consists of three main components (Fig.1):

- Client-side: a part of the application that handles user requests and interacts with Ansible playbooks via API.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

- Configuration management scripts module: Ansible playbooks for setting and applying specified configuration to target system.
- Target infrastructure system (server, virtual machine or PC with already installed operation system, etc.)

There are many approaches to end-to-end testing and some suggest that testing should be done in three stages with each component tested separately. Often, different engineers with different knowledge, skills, and competencies conduct each stage of testing. However, this approach has some disadvantages. Firstly, it can take a considerable amount of time to complete all three stages of testing, especially if the gap between each stage is long. This can lead to delays in the development process and make it more challenging to fix any issues that are identified.

Another issue with this approach is that it can be challenging to identify the root cause of any issues that arise. If an error is identified in one component, it can be challenging to determine whether the issue is specific to that component or if it is related to another part of the system. This can lead to a significant amount of time spent on troubleshooting and can further delay the development process.

Moreover, the three-stage approach to end-to-end testing may not capture all the possible interactions and dependencies between the different components of the system. This can result in issues being missed, which can lead to unexpected behavior when the system is deployed in a production environment. Additionally, this approach can be costly since it requires a significant amount of time and resources to execute and maintain.

Therefore, for end-to-end testing of the considered application an approach is needed that will not break the system into separate components but will consider the entire system as a whole (Fig.2).

Thus, minimizing the time complexity during the process of end-to-end testing is crucial. Moreover, the entire testing process can be carried out and monitored by a single engineer.

End-to-end testing may encounter several issues, including complexity, time-consuming nature, difficulty in reproducing errors, inconsistency, debugging challenges, high cost, and so on.

Complexity is one of the primary challenges of end-to-end testing, especially when testing complex systems. Testing multiple components of such systems makes the testing process challenging. Tests may also consume a lot of time, making frequent testing difficult [3].

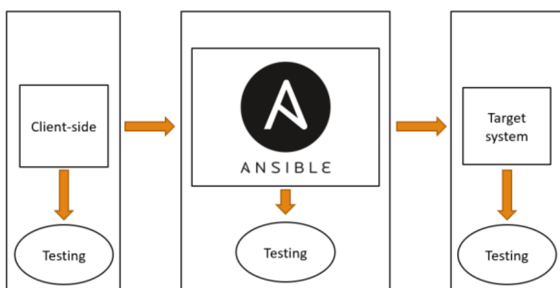

Fig. 1. The main components of the application under test

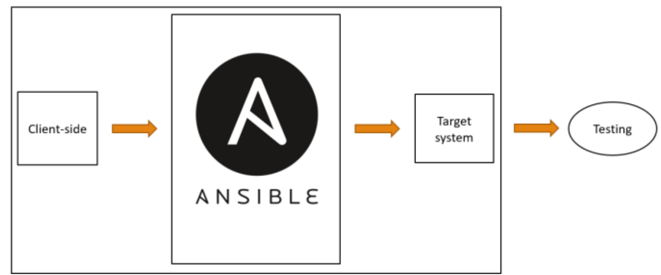

Fig. 2. Approach to end-to-end testing of the considered application

Another problem that may arise during end-to-end testing is difficulty in reproducing errors. End-to-end testing can make it difficult to identify the cause of an error when it arises due to testing the entire system. Furthermore, end-to-end tests can be inconsistent, meaning they may fail repeatedly for various reasons. This can be caused by different factors such as network issues, browser version mismatch, and race conditions.

Debugging end-to-end tests can indeed be a challenging task, especially when a test fails. Identifying the root cause of the failure can be problematic, as the issue may be located in any of the various components of the system. This can make it difficult to isolate the source of the problem and resolve it. As a result, engineers may spend a significant amount of time diagnosing and fixing issues, which can increase the overall development time.

Moreover, end-to-end testing requires a considerable amount of hardware and software resources. This is because the testing process involves the comprehensive evaluation of the system from the user interface to the database, which requires a substantial amount of computational power.

Maintaining end-to-end tests can also be challenging over time, especially when the system being tested evolves and changes. In complex systems, end-to-end testing may not cover all possible scenarios, leading to limited coverage. Tests may also provide insufficient feedback to developers as it may be challenging to pinpoint the exact location of a problem in the system.

Finally, end-to-end testing can be too costly. Conducting end-to-end tests may require the use of a large number of hardware and software resources as well as the need for an adequate number of qualified specialists to write and maintain tests.

Overall, end-to-end testing has its advantages and disadvantages, and its effectiveness depends on several factors, including system complexity, the number of resources available for testing, and the experience of the engineering team. Understanding these issues can help developers and testers create a more effective testing strategy to ensure system quality and reliability.

When developing software and tests, one of the generally accepted methodologies is used:

- TDD (Test Driven Development) [4] is a software development methodology that is based on repeating short development cycles: initially, a test is written that covers the desired change, then program code is written that implements the desired behavior of the system and allows the written test to pass. Then, the

written code is refactored with constant checking of the passing of tests.

- TDD (Type Driven Development) [5] is based on types. In this case, data types and type signatures serve as a specification for the program. Types also serve as a form of documentation that is guaranteed to be updated.
- BDD (Behavior Driven Development) [6] involves describing user scenarios in natural language by testers or analysts.
- DDD (Domain Driven Design) [7] is a set of rules that allow for making the right design decisions. This approach significantly speeds up the process of designing software in an unfamiliar domain.
- FDD (Features Driven Development) [8] attempts to combine the most recognized software development methodologies in the industry, based on important functionality (properties) of the developed software for the customer. The main goal of this methodology is to systematically develop real, working software within the set deadlines.

It was decided to use BDD, as it is the only approach that involves obtaining natural language test documentation as output. Additionally, this methodology allows a single engineer (tester) to independently carry out the entire end-to-end testing cycle, from writing test scenarios to creating a report on the results of testing. This solves one of the main problems - involving multiple engineers with different sets of knowledge and competencies in testing.

When conducting end-to-end testing, specialized tools are usually used for this purpose. Of course, there are more or less universal tools, but it should be understood that the intricacies of end-to-end testing may differ even among software systems operating in the same domain. Therefore, for each system, its own approach and tool for end-to-end testing are usually developed. Moreover, implementation may vary depending on the skills or preferences of engineers, as well as the specifics of the software. This can be a full-fledged application with a graphical interface, a console program, a framework that is embedded in the project with automated tests, and so on.

For example, within the scope of end-to-end testing of V2X (Vehicle-to-Everything) systems [3], a distributed application is used, each node of which emulates a real microcontroller. Moreover, it is necessary to emulate not only the "hardware", but also a specialized message exchange protocol. V2X is a technology that provides communication between vehicles and other objects, such as infrastructure and pedestrians. It should be understood that the criticality of an error in such a system can cost someone's life. Therefore, in such systems, testing tools must be as close as possible to the real environment: using a minimum of mock objects, distributed, well supported and updated. All these factors make end-to-end testing tools very heavy for development, support, and financing, so such tools should only be used when there is an urgent need. Otherwise, resources spent on development and support will be wasted.

Speaking of universal cross-functional testing tools, the most well-known ones are Jaeger and Zipkin [9]. These two very similar tools provide functionality for fairly detailed identification of system failures in a software system. These tools have many advantages: they are easy to set up thanks to detailed documentation and ease of use, they are suitable for almost any software system, provide a wide range of functionality and a graphical interface, which means that there is no need to spend time developing and supporting this functionality. However, the universality of these tools also has its drawbacks. Jaeger and Zipkin do not provide functionality for executing test scenarios, i.e., testing will still have to be done manually, but the tools will help to identify the location of the error more easily and quickly. The results of the tools' work may be difficult for non-technical people (such as management) to understand, so all reports will also have to be written manually. Also, both tools are quite "heavy", which may require additional hardware resources for deployment. Jaeger and Zipkin do not support all programming languages, so if the system is written in a specific language, it may simply not be possible to configure the tools. Finally, these two tools may pose certain risks in terms of information security: if access is configured incorrectly or improperly, various sensitive information may leak. Thus, Jaeger and Zipkin are not without drawbacks, but are still good tools for cross-functional testing, but only as additional control systems. Nevertheless, the functionality does not allow for complete control of cross-functional testing only through these tools.

One of the most popular cross-functional testing tools today is a software tool in the form of a framework for a programming language on which engineers write automated tests. This approach is relevant not only for applications that perform mobile computing [10], but also for other systems from completely different subject areas. The idea of implementing such frameworks is as follows: for a test framework of some programming language (e.g., the Pytest framework for the Python language), a wrapper is written that allows desired actions to be performed with the system under test. Creating a cross-functional testing software framework can offer several advantages:

- Flexible customization. With a custom testing platform, one can adapt their testing to the specific needs of their project, providing greater flexibility and control over the testing process.
- Reusability. A custom framework can be reused in multiple projects or teams, saving time and effort in the long run.
- Integration opportunities. Integrating a custom testing platform with other tools and systems can help simplify and automate the testing process.
- Cost savings. Creating a custom framework may be cheaper than purchasing, learning, and implementing a commercial testing tool or framework.
- Learning opportunities. Creating a custom framework provides opportunities for one's team to learn as they develop and refine their skills in software development and testing.

III. THE PROPOSED APPROACH

To arrive at the picture shown in Fig.2, an approach to end-to-end testing was developed that can be applied by a single engineer and minimizes downtime between testing of system components.

Behavior Driven Development (BDD) [11] methodology is used for end-to-end testing of both individual components and the system as a whole. BDD is a development approach based on behavior description, where an engineer writes descriptions such as "As a user, when I click the Start button, the menu should be displayed as shown in the picture." Classic development with tests follows. BDD involves engineers describing user scenarios in natural language. Thus, the output will include not only end-to-end test scenarios and their results, but also natural language descriptions that can be used for documentation or reporting. Moreover, a person who is not familiar with the technical implementation of the system can understand the results because all results will be described in natural language.

One of the main advantages of BDD is that this approach focuses on business needs rather than technical implementation details. Tests are written in natural language that everyone can understand, which improves communication and minimizes possible misunderstandings. This approach allows customers and developers to determine what needs to be tested without additional costs for translating technical documentation.

BDD also leads to improved test coverage of the product because this methodology aims to cover all possible application usage scenarios. Based on these tests, potential issues can be easily identified before they become real problems for users.

Furthermore, BDD enables the creation of tests that are easier to maintain in the future. This is because changes in the application require changes in all related tests, so the easier the tests are to maintain and modify, the easier it is to make changes to the application itself.

However, the BDD methodology has its own limitations. BDD can be quite difficult to understand, especially for new developers and testers. It may require them to learn new terminology and testing approaches.

BDD may require additional effort during development to create functional requirements that can be used to write tests. This can take additional time.

Additionally, BDD may require workers to put in significant effort to understand the behavior of the application. This may require a lot of training, practice, and time, which may be unrealistic for some companies at present.

Integration with tools can be challenging, especially with multiple tools at once, to run BDD tests. This may require additional costs to maintain tool settings up-to-date.

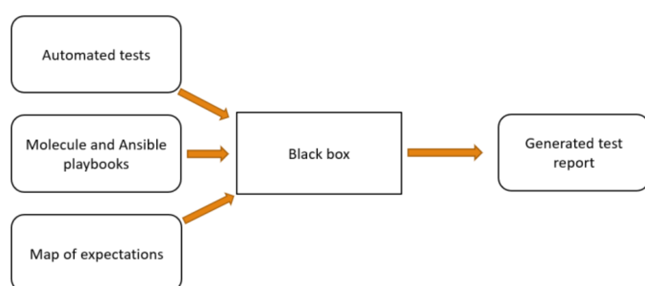

Fig. 3. System-level design

However, the main problem with applying BDD in end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure is the presence of components such as configuration management script modules and target infrastructure systems.

In the classical application of BDD, it is suggested to cover end-to-end tests only for the client-side, which is written in commonly used programming languages such as Java, C#, Python, etc. However, it is unclear how to deal with configuration management tools (Ansible/Puppet) and even more unclear how to apply BDD to end-to-end testing of server or virtual machine states. To address this issue, a software tool was developed, which is a Python-based framework that allows the application of BDD to specific components for this approach.

IV. Implementation

IT companies develop specialized frameworks (including for testing) to address various tasks within their systems, taking into account the specifics of the product and providing the necessary tools for interacting with it. Since automated testing of the client-side is implemented using the Python framework Pytest, a software tool was developed that is a framework for the Python and allows the application of BDD to specific components for this approach.

Functional requirements for the developed software:

- The software shall prepare the infrastructure on which end-to-end testing will be performed (in this case, creating virtual machines), and after completion, release resources (i.e., delete virtual machines).
- The software shall provide the ability to test the client-side of the application.
- The software shall provide the ability to test configuration management scripts.
- The software shall provide the ability to test the state of the target system after applying the necessary configuration.
- The software shall generate a summary report on the results of end-to-end testing in natural language.

The system-level design [12] of the developed software can be described by the diagram shown in Fig.3.

The developed framework is essentially a versatile black box that can take in automated tests, Ansible playbooks, Molecule scripts, and a map of expectations as inputs, all written in Python. These inputs are then used to execute the end-to-end tests on the system under test (SUT). Testing in this case means, for example, checking the correctness of the scenario for installing security updates on a group of virtual machines.

One of the key advantages of this framework is that it is easily customizable and extensible, allowing users to tailor it to their specific testing requirements. Users can add their own test cases and test scripts, as well as customize the test environment and configuration to suit their needs.

Upon execution, the framework automatically generates a test report in Allure, a flexible and open-source platform for test reporting. The report includes comprehensive

descriptions of the test cases in natural language, making it easy for users to understand and interpret the test results.

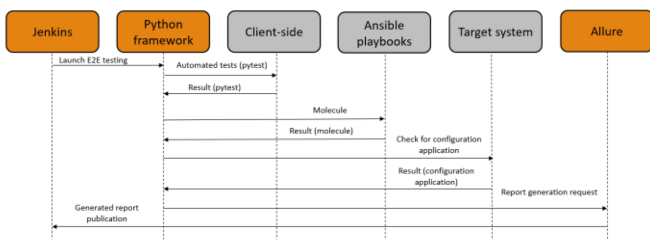

Fig. 4. UML sequence diagram

The software solution developed for this approach is written in Python, with a specially configured Jenkins job as the user interface. The Pytest and Molecule tools are responsible for end-to-end testing of the entire system, on which a module for applying the BDD approach to end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure was created.

The developed software solution is highly flexible and versatile framework for Python, which can be easily integrated into any project with automated end-to-end testing requirements. It provides a seamless and well-structured interface for writing and executing end-to-end tests, using a variety of inputs such as autotests, Ansible playbooks, Molecule scripts, and a map of expectations.

To help visualize the inner workings of the end-to-end testing system, a UML sequence diagram has been developed, which captures the interrelation of all components involved in the testing process (Fig.4). This diagram illustrates the flow of events that take place during the execution of the end-to-end tests, starting with the initialization of the test environment and ending with the generation of the test report.

The diagram shows how each component of the system interacts with the others, providing a clear and concise overview of the entire testing process. It highlights the critical role played by the developed framework in managing the end-to-end testing process, as well as the importance of other components such as the Ansible playbooks and Molecule scripts in setting up the test environment and ensuring the correct behavior of the system under test.

The primary modules of the application for managing the configuration of enterprise virtual infrastructure are highlighted in gray, while the modules of the developed software solution are highlighted in orange. Jenkins provides the user interface through which the engineer configures and launches the end-to-end testing process. Subsequently, the Python framework executes the test scenarios sequentially for each module: the client-side (using Pytest), Ansible scripts (using Molecule), and the target system (checking its initial and final states). The results are aggregated into a report; which Allure generates in natural language. The final report is published in Jenkins, where reports for each of the end-to-end testing process runs are stored. The interrelation of all components of the final end-to-end testing system can be described by a UML sequence diagram.

The architecture is shown in Fig.5.

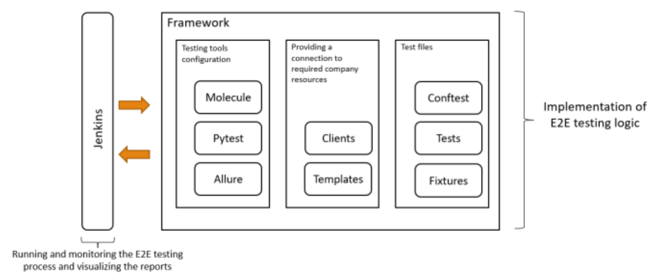

Fig. 5. High-level design of the developed software

The framework comprises three primary components:

- A module for preparing test data and deploying the test infrastructure, which includes fixtures - objects that can be considered a set of conditions required for the test to run. For example, fixtures are often created to generate data before the test starts and return it for use in the test or before the test. This module also handles the preparation of the test infrastructure (creating virtual machines) and the preparation of test data such as IP addresses of virtual machines, login credentials, operating system names, etc.

- The module that implements methods for accessing the necessary services of the company during testing. This module creates conditions for testing that replicate the production environment: to correctly use virtual machines created in the previous module, they shall be properly registered with third-party services developed by other teams. Additionally, this module is responsible for verifying the correctness of the end-to-end testing process with respect to other systems within the company (answering the question "Did our system break another team's system?").

- The module for configuring testing tools and reporting. This module implements the configuration of the main testing tools: Pytest for the client-side, Molecule for configuration management, and shell scripts for checking the state of virtual machines. It also generates a summary report of the test results in natural language and loads it into Jenkins for visualization.

The Jenkins job plays a crucial role in implementing the user interface for starting the end-to-end testing process and visualizing the results, as shown in Fig.6. It serves as a platform for managing the entire testing process, from the initial setup and configuration of the testing environment to the execution of tests and the generation of reports.

Through the Jenkins interface, users can initiate the testing process and monitor its progress in real-time, enabling them to quickly identify and address any issues that may arise. Additionally, Jenkins can be configured to automatically trigger tests based on predefined conditions or events, further streamlining the testing process and reducing the likelihood of errors or oversights.

The visualization of results in the Jenkins interface provides a clear and concise overview of the testing process and its outcomes. Users can easily interpret the results and identify any areas of concern, allowing them to take prompt corrective action and ensure the software meets the necessary quality standards.

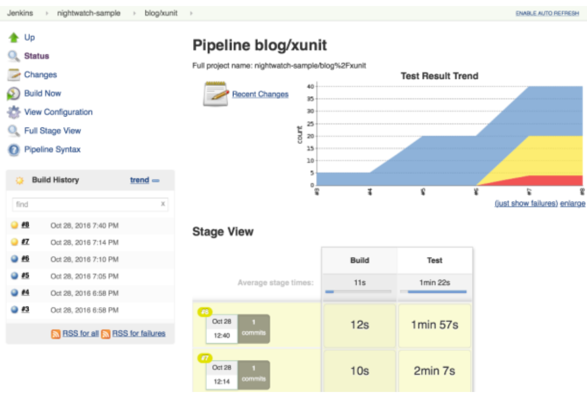

Fig. 6. Jenkins job

Through the Jenkins functionality, it is possible to configure the parameters of end-to-end testing: determine the number of virtual machines, select desired operating systems, configure the system under test, etc.

A summary format displays the last few runs of end-to-end testing, whether they were successful or not, and if not - at what stage an error occurred. Additionally, it is possible to evaluate a graph of the number of successful, skipped, and unsuccessful test scenarios.

The final generated test report can be also accessed through the Jenkins job interface and consists of a set of test scenarios with various input data described in formal language steps, expected results, and the success of executing a particular scenario (Fig.7).

V. RESULTS

An end-to-end testing approach for the application for managing the configuration of enterprise virtual infrastructure was developed, and a Python framework with a Jenkins user interface and natural language test report generation in Allure was created to implement and apply this approach.

The metric chosen to evaluate the usefulness of the new cross-functional testing approach was the time required to execute a single test run on operating system update functionality: checking correct credentials uploading, correct operating system update results and correct deletion of temperory files after update. Ten experiments were conducted using the developed framework, and ten experiments were conducted using the old approach (with step-by-step testing of each module). It should be noted that downtime between stages in the old approach was not included in the metric, i.e., only the time of the testing process was taken into account.

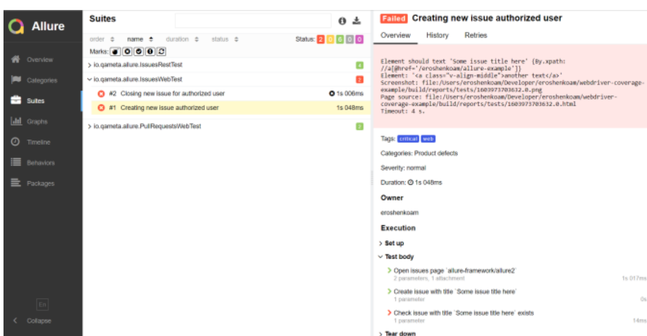

Fig. 7. Test report

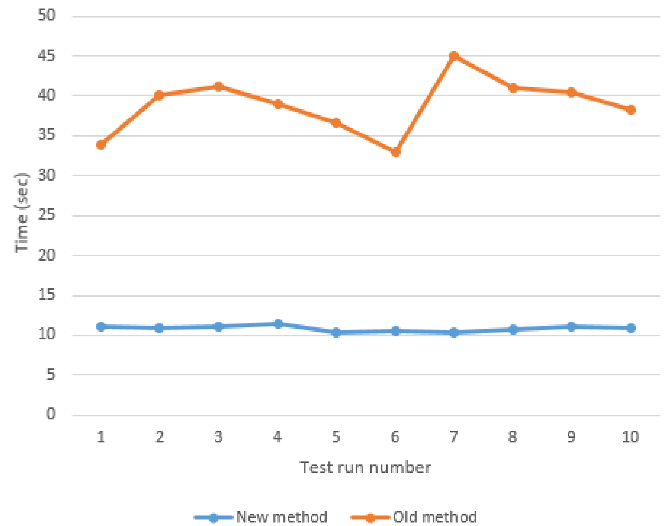

Fig. 8. Results of the first experiment

Initially, an infrastructure consisting of one virtual machine was used. On average, using the developed framework requires 10–11 seconds per test run, while the old approach requires 30 to 45 seconds (Fig. 8).

Next, the size of the infrastructure was increased to ten virtual machines, and the same experiment was conducted. The difference was more significant. The time required for a single test run using the developed framework remained at 10–11 seconds. However, the old approach showed a much longer execution time for a single test run - from 200 to 250 seconds (Fig. 9).

The results obtained from the evaluation of the new approach to end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure were statistically significant and can be explained through the principle of parallel operations. The use of the developed Python framework and its support for parallelism led to a significant reduction in the time required to execute a single test run compared to the old approach with step-by-step testing of each module.

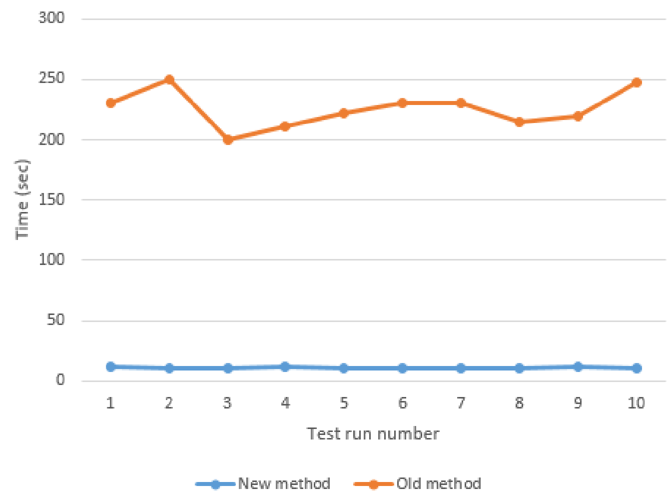

Fig. 9. Results of the second experiment

The statistical analysis of the experimental results using a t-test showed a significant difference ($p < 0.01$) between the mean execution times of the old and new approaches. This indicates that the new approach is superior in terms of efficiency, as it enables parallel operations and thereby reduces the time required for end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure.

The principle of parallel operations is widely recognized in computer science and engineering as an effective means of improving the efficiency of various computing tasks. It involves breaking down a large task (testing the whole virtual environment) into smaller subtasks (testing each node) and executing them concurrently, thereby reducing the overall time required for completion. In the case of end-to-end testing of the application for managing the configuration of enterprise virtual infrastructure, the use of parallelism is particularly beneficial as it allows for the testing of multiple virtual machines simultaneously, resulting in significant time savings.

It is worth noting that with the new approach to end-to-end testing, the entire process is performed and controlled by a single engineer, as opposed to the old approach, which required the involvement of three different engineers with varying levels of knowledge and expertise. This allows for a more streamlined and efficient testing process, as the same engineer is responsible for each stage of the testing process and can quickly identify and resolve issues as they arise.

During the testing process, three engineers are, of course, more efficient than one, since they can confer with each other and find the root cause of the problem faster. But in this case, these are engineers from different teams who perform rather isolated parts of a single process. Therefore, if one of the engineers had a problem, the others could not help him much, because they did not know the subject area of the other teams well. Therefore, the new framework gives the best result in such a situation.

Additionally, the developed software framework for end-to-end testing generates a detailed report in natural language, which provides a comprehensive overview of the testing process and the results obtained. This report is automatically generated by the software, eliminating the need for manual report writing, which was necessary with the old approach. This saves considerable time and effort, allowing engineers to focus on other critical aspects of the development process.

Furthermore, the use of natural language in the generated report makes it easy for stakeholders to understand the testing results and make informed decisions regarding the software development. This can be particularly useful for managers and other non-technical team members who may not have the technical expertise required to interpret traditional testing reports.

VI. Conclusion

The proposed approach to end-to-end testing for the application for managing the configuration of enterprise virtual infrastructure offers several advantages over the traditional approach, including a more streamlined testing process and the automatic generation of a comprehensive report in natural language. These benefits can help engineers save time and effort while improving the quality of the software and ensuring that it meets the required standards and specifications.

References

[1] W. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in Proceedings of IEEE COMPSAC, 2001, pp. 166-171.

[2] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery," in 2021 IEEE International Conference on Big Data and Smart Computing (BigComp), Feb. 2021, pp. 16-23.

[3] J. Wang, Y. Shao, Y. Ge, and R. Yu, "A Survey of Vehicle to Everything (V2X) Testing," Sensors, vol. 19, no. 2, p. 334, Jan. 2019.

[4] K. Beck, "Test Driven Development By Example," Addison-Wesley Professional, Jun. 2002.

[5] E. Brady, "Type-Driven Development with Idris," Manning Publications, 2017.

[6] J. F. Smart, "BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle," IEEE Software, vol. 36, no. 4, pp. 101-102, Jul./Aug. 2019.

[7] H. Wesenberg and J. Olmheim, "Agile Enterprise Software Development Using Domain-Driven Design," Conference on Object Oriented Programming Systems Languages and Applications, 2007.

[8] A.F. Chowdhury and M.N. Huda, "Comparison between Adaptive Software Development and Feature Driven Development," International Journal of Computer Applications , vol. 60, no. 6, pp. 37-42, Dec. 2012.

[9] C. Martinez Hernandez, A. Martinez, and A. Rojas, "Comparison of End-to-End Testing Tools for Microservices: A Case Study," in 2019 IEEE 43rd Annual Computer Software and Applications Conference, 2019, pp. 539-544. DOI: 10.1109/COMPSAC.2019.00114

[10] I. Satoh, "A Testing Framework for Mobile Computing Software," IEEE Transactions on Software Engineering, vol. 29, no. 12, pp. 1112-1121, Dec. 2003.

[11] Testlio, "What is BDD (Behaviour-Driven Development)? - Software Testing," Testlio website. Available: https://testlio.com/blog/what-is-bdd-behavior-driven-development/. [Accessed: May. 10, 2023].

[12] S. Knight and R. Eggert, "The Role of System-Level Design in the Development of Autonomous Systems," Journal of Systems and Software, 2021. DOI: 10.1016/j.jss.2021.111303.

[13] Jenkins Plugins, "Allure Jenkins Plugin," [Online]. Available: https://plugins.jenkins.io/allure-jenkins-plugin/. [Accessed: Mar. 28, 2023].