Software testing automation within master-agent architecture

Bojana Ivanovic, Vanja Mijatov and Branko Milosavljevic
Faculty of Technical sciences, Novi Sad, Serbia
bojana.ivanovic@uns.ac.rs, vanja.mijatov@uns.ac.rs, mbranko@uns.ac.rs

***Abstract*—Paper describes a test automation system modeled through the master-agent architecture, where the master node defines tasks as a part of the testing workflow and delegates them to the agent node. Agent node is the one that does the actual work, which is running the tests. After finishing the test session, agent generates descriptive report about the executed test session and sends it back to the master node, where it is available to the end user. The agent part of the system is implemented within containerized and controlled environment. In addition to the implementation details, the observed shortcomings that led to the expansion of the system were highlighted with suggestions for possible future expansion of the system.**

Keywords: test automation, master-agent architecture, Jenkins, Allure, Docker, pytest

I. INTRODUCTION

Automation process, in general, brings great advantages over manual labor, which are: increased productivity, more efficient resource usage, higher product quality and lesser probability of human errors. When it comes to information technologies, automation is of crucial importance, primarily because of the great interest in reducing the probability of error made by humans involved in the process, but also increased efficiency and all other benefits of automation over manual work.

As a paradigm, automation has made great changes in modern software development. Whether that software is a part of the hydropower plants system, ships, aircrafts or booking systems, we expect them all to work flawlessly. In practice, it is almost impossible to write a bug-free software, so previously noted expectations are unrealistic. It is possible though to create an adequate software by reducing the number of problems in it to a minimum. This is done by testing the software. Testing is a mandatory stage in software development process and perhaps presents most challenging and responsible task for the developers. Testing brings problems to the surface, so that by eliminating them we can be one step closer to the "almost perfect" software.

The modern age in software development has led to situations where teams are working on software development under constant deadlines for delivery of the software as well as specific requirements by customers. What regularly happens is that the development of key functionalities of a software is delayed or completely excluded from the project in order to meet required deadlines. What teams and project managers face every day is that, in addition to meeting deadlines, software needs to be developed with minimal consumption of resources (primarily money). It also happens that companies want to adequately test the software as quickly as possible but also thoroughly. In order to achieve that goal, it is necessary to turn to the automated way of testing. [1]

There are many definitions of test automation and it is often confused with the software being tested. In general, test automation consists of several activities, which vary but they usually include development and execution of the test scripts, verification of testing requirements and use of automated testing tools [2]. Moreover, it can be defined as a usage of special software in order to control tests that are being executed, as well as for comparison of the test results with predicted values [3][4]. As previously mentioned, automation process brings great advantages when properly implemented.

Nowadays, test automation as a concept is gaining great popularity due to significantly increased need for quality software that works without major issues and downtime. By doing test automation, test sessions can be executed more often on different machines which results in earlier detection of potential issues. This study is motivated by the idea of automated testing and creating a system for it that can also be easily delivered to the customer. The starting point was the master-agent architecture, where master node delegates the job to one or more agent nodes that would actually do the work defined and delegated by the master node. Motivation was widened by the idea that several existing tools could be combined into one solution in order to enable the full cycle of automated testing and are able to run complex testing processes on a single or multiple devices.

II. METHODOLOGY

Master-agent architecture was first developed for database replication but nowadays it is used in distributed systems for communication between nodes where one of them has greater power over the others. In that relationship master node is the one that delegates tasks to the agent nodes. Automated testing system based on master-agent architecture consists of a master node located in the cloud, while the agent node is a physical machine (computer) and this is achieved by using the Jenkins CI/CD tool [5]. Jenkins is used for automation of all kinds of tasks related to building, testing, delivering or using the software, facilitating continuous integration and delivery of the software (CI/CD) [6]. Communication between nodes is established via TCP/IP protocol. In this research it was opted for Jenkins because of familiarity with the tool but also because of its flexible behavior. By default, Jenkins comes with a limited set of features but also provides hundreds of free plugins for customization

of the project and they are developed continuously. These extensions cover various aspects of software development from version control, build tools and metrics for measuring quality of the code [7] and that is one of the main reasons why this tool was selected for this project. The flexible behavior is one of the reasons why Jenkins is often chosen as a CI/CD tool because it allows creating complex pipelines [8]. For any customization, master node needs to be additionally configured by adding one or more of these plugins.

A containerized application is placed on the computer (agent node) that executes the testing process. Containerization was chosen because of the need for automated testing to be performed in a strict and controlled environment that can be easily configured so that there is no need for additional configuration on the host machine. Applications packaged inside of a container can operate on any machine that supports chosen containerization technology without any modification, because when containerized they do not depend on any machine's configuration [9]. One of the prerequisites for testing was controlled environment using containerization because it is fully configurable and isolated so that the host machine does not interfere with the testing process. For these purposes was used Docker [10]. Moreover, Docker gives full control on dependencies used in the environment, having just the minimum requirements of the operating system or having only needed libraries and tools, making the final product as lightweight as possible [11]. Additionally, because of the containerization this kind of a setup is easily distributed to the users. This tool was chosen among others because of its widespread use and large and dedicated community.

Allure framework [12] is a reporting tool that provides systematic and intuitive representation of test results in form of a report. After a test run, Allure report will contain all relevant details about the test session and it will provide enough technical details for developers in a more descriptive way. The problem with Allure reports is that the size of the report increases with the number of executed tests. In addition, this is also a problem because Jenkins is not conceived as a tool for long-term storing of reports with high disk consumption. Overcoming these drawbacks was one of the ideas which arose during early stages of the research and finding solution to them was also covered in later stages of the research.

III. SYSTEM DESIGN

System that is proposed in this paper is based on master-agent architecture and implemented using several existing tools. Proposed system is a generic solution, where any of the chosen tools can be substituted with other tools with the same purpose.

The idea was that the whole testing process flow starts with the master node which delegates testing tasks to the agent node. The actual testing is being performed on the agent node machine, while the master node orchestrates testing tasks, collects results and generates and stores report of executed test session. Before hence, agent needs to download the tests which are stored on the Git repository. Once testing is completed, Allure report is generated as an output and it is forwarded back to the master node where it can be previewed by user at any time.

Creating of an agent node is done through the Jenkins application by setting up basic node information. What stands out is the information needed for functioning of the agent, such as directory used as workspace and mode in which node will be used (in this case used to the maximum whenever available). Configuration regarding initial setup of the agent depends on the manner of connecting master to the agent. In this particular case, option that was selected was connecting agent to the master directly due to lesser need for configuration. In addition, every agent needs tools needed for the execution of the given task. Agent node created in this implementation has Git in order to be able to pull the tests stored on the repository (which needs to be configured in the Jenkins master application and provided to the agent) and Allure plugin which needs to be configured globally inside of the Jenkins master application.

Initial idea was to define a job inside of the Jenkins application in order to initiate the test run and to have the Allure report as the output. Moreover, the intention was to have the tests stored on the Git repository and not locally on the machine so one of the first steps of the job was to download the tests from the repository. The goal was also to increase flexibility by not having tests on the agent machine prior to the test run, because in that way we can always pull the newest tests (if there were any changes) or the job would pull all or just specific group of tests needed for the test run.

The job itself can be defined in several different ways. The first one that was tried was the *Pipeline* mechanism but the idea was quickly abandoned. The downside of this approach was the script for job definition. It grows and gets complicated over time and even though it is a programming code, it becomes difficult to maintain. Additionally, the requirement that tests need to be first downloaded could not be fully fulfilled in the Pipeline mechanism due to Git tool not being able to be effortlessly integrated into.

After having an insight in the drawbacks of the Pipeline mechanism, the solution chosen for job definition was the *Freestyle project* which had wider set of options. Problem of using version control tools was solved by using this type of job definition. In this case further configuration was needed in the form of specifying repository location and authentication method for repository access. The other sections of job definition like command for starting the test session and location of a directory for storing reports were identical in both *Pipeline* and *Freestyle* job definition.

As one of the key requirements was that the system for test automation works inside of a container, it was necessary to allow Jenkins to work inside of the container. In order for a node to have master behavior, it is necessary for it to contain the entire Jenkins application so that it could provide all the required functionalities. This is possible if Jenkins master application is packed within a Docker in two ways: using Jenkins as base image or by packaging Jenkins master application (*jenkins.war*) inside of a container. In order for a node to behave as an agent, it is necessary to provide agent behavior by packaging agent.jar within Docker container. Agent.jar is a java application that connects agent node to the master in cloud and allows executing commands that are orchestrated by the master node. For the needs of the implemented solution two Docker images were created. First one with

the behavior of both master and an agent node (node contains both jenkins.war and agent.jar). Second image was created to behave like an agent. Both of them were used during the evaluation of the implemented solution.

In the early stages of the implementation the goal was to execute tests using *pytest* [13] framework and each test was based on few simple claims (assertations) which served as a test run and a proof of concept that the whole system flow was working. After confirming that the full cycle was operable, system needed to be validated for a specific software. Initial idea was to create a controlled testing environment that works within master-agent architecture for a specific software by *Typhoon HIL Inc*., called *Typhoon HIL Control Centre* (THCC) [14] and their *TyphoonTest* [15] library, library is based on *pytest* which was used in the initial phase of the development. *TyphoonTest* is used by the company in order to test real-time systems. To test systems in the real time, there is cooperation with the simulator where the test will be practically performed and return the result. T*yphoon-allure* was used for generating the Allure report and it is a part of the THCC and represents the wrapper around the Allure framework itself. Since one of the main ideas was to evaluate the system with the Typhoon HIL domain software, their control center is packed inside of a container that represents the agent node. Figure 1. shows structure of the agent node which is configured to use *Git* tool for pulling the tests from the repository before testing, the *Allure* framework so it could generate report upon testing and THCC for the testing. Also, left part of the Figure 1 shows configuration of the master node in the cloud.

During the development and testing phases, we identified few drawbacks of the system:

1. User could not see any progress during the test run;
2. Test session report is available only on the master node;
3. Disk space usage problem.

The first drawback is a user experience problem, where during the testing process user has no knowledge about the test session being executed or about any progress of the testing process in the runtime. This drawback is something that needs to be addressed in further

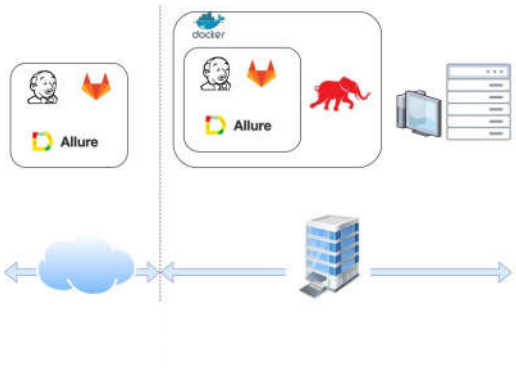

Figure 2. Proposed solution with the cloud part of the system on the left side and agent node on the right side

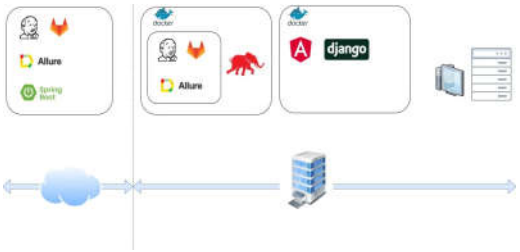

Figure 1. Enhanced solution with the cloud part of the system on the left side and agent node on the right side

development since whole testing process can have significant duration (e.g. couple of hours) and during that time test process is a black box for the end user. Once the test session is completed, generated Allure report will be forwarded to the master node and it would be accessible from the Jenkins application. Jenkins master application is usually not available to all users but only perhaps to the team lead or QA team, which represents second drawback. Third problem, regarding disk space consumption is a very important one since the generated Allure reports occupy a lot of space on disk. Every Allure report consist out of a significant number of JSON files so their long-term storing is challenging.

Due to previously mentioned drawbacks of our first solution, the entire system has been expanded by creating another Docker container containing two applications: *Django* application that monitors testing processes and *Angular* application that displays previous results and provides better user experience to the end user. Once the testing process is completed, *Django* application packages generated Allure results inside of a zip archive and sends the archive to the Spring boot application. Left part of the Figure 2. shows cloud part of the system, where previously mentioned *Spring boot* application was placed. It works with test session data stored inside of a *MySQL* relational database and stores all the reports. Instead of Jenkins master application, Spring boot application is the one that stores the reports. Still, the user does not have an insight into real-time results which is still a drawback but has preview of all previous sessions and access to Allure report for each session.

Apart from the preview of the test session data in runtime and further system evolving with regards to disk space consumption problem, one of the next steps would be a better preview of the data in the form of dashboard with graphical preview of test results for all test sessions and most recent test session with filtering options which could enable preview of data within a given date range or by specific criteria.

IV. CONCLUSION

It is believed that test automation provides developers more time and that it is superior to manual testing. Unfortunately, both of those statements are false. Firstly, test automation will just provide developers more time to focus on more complicated issues. Secondly, automated and manual testing both have advantages so the first one can never completely substitute the second. In the era of modern development, test automation has becoming more and more important because of long-term benefits for both developers and organizations. Best part about it is wide

set of options when it comes to tool selection for its implementation. This paper presented just one solution for test automation, where the system was designed as master-agent architecture.

Future development of proposed solution consists primarily of dealing with described drawbacks of the system. Since proposed solution needs to store significant number of Allure reports, one of the next steps would also be addressing further the disk space usage problem by analyzing it from the big data point of view. Furthermore, one of the open research questions is to come up with a more efficient way for their search since Allure reports are multimedia documents that contain text and images (visualized data via charts).

REFERENCES

[1] Dustin, Elfriede, Jeff Rashka, and John Paul. Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance. Addison-Wesley Professional, 1999.

[2] Taipale, Ossi, et al. "Trade-off between automated and manual software testing." International Journal of System Assurance Engineering and Management 2.2 (2011): 114-125.

[3] Garousi, Vahid, and Mika V. Mäntylä. "When and what to automate in software testing? A multi-vocal literature review." Information and Software Technology 76 (2016): 92-117.

[4] Huizinga, Dorota, and Adam Kolawa. Automated defect prevention: best practices in software management. John Wiley & Sons, 2007.

[5] https://www.jenkins.io/ , Accessed May 2022

[6] Hembrink, J., and P. G. Stenberg. "Continuous integration with jenkins." Coaching of Programming Teams (2013).

[7] Smart, John Ferguson. Jenkins: The Definitive Guide: Continuous Integration for the Masses. " O'Reilly Media, Inc.", 2011.

[8] Armenise, Valentina. "Continuous delivery with Jenkins: Jenkins solutions to implement continuous delivery." 2015 IEEE/ACM 3rd International Workshop on Release Engineering. IEEE, 2015.

[9] Bui, Thanh. "Analysis of docker security." arXiv preprint arXiv:1501.02967 (2015).

[10] https://www.docker.com/ Accessed May 2022

[11] Vase, Tuomas. "Advantages of docker." (2015).

[12] https://docs.qameta.io/allure/ Accessed May 2022

[13] https://docs.pytest.org/en/7.1.x/ Accessed May 2022

[14] https://www.typhoon-hil.com/products/hil-software/ Accessed May 2022

[15] https://www.typhoon-hil.com/products/typhoon-test-ide/ Accessed May 2022