Is a Software Quality Pipeline realizable on microcomputers?

Sebastian Wientzek, Tim Burkhardt, Nestor Patient Tchamba Sefekme, André Konhardt
Institute of Computer Science
Hochschule Ruhr West, University of Applied Sciences
Bottrop, Germany
{ sebastian.wientzek, tim.burkhardt, nestor-patient.tchamba-sefekme, andre.konhardt } @stud.hs-ruhrwest.de

Abstract – The challenge of continuous integration and continuous delivery of quality software still discourages many companies today from setting up and using a software quality stack. In particular, setting up the individual tools, such as source code management, code integration or the deployment of images and connecting them to form a complete system, is initially associated with greater effort. For this reason, this paper examines the extent to which such a software quality stack can be mapped as a cluster on single-board computers. The effort involved in setting up the cluster will be examined in more detail and evaluated in terms of performance using a small web application. The special feature described in this paper is to map the software quality stack as a distributed system on different Raspberry Pi's.

1 Introduction

Software quality stacks are a convenient way for companies to test and publish software updates and new projects with little effort. For this purpose, the various tools are connected with each other and expanded to form a holistic system. It's also used as a platform for software developers to evaluate the quality of the source code and to search errors within the software or especially the source code.

Modern and established software companies are in constant competition with each other and want to publish their software faster and easier. The requirements in software development are constantly growing, so that an ever-increasing degree of automation is necessary to correct software errors on time and with little effort. The setup and installation of such a software quality stack is usually associated with greater effort and requires not only monetary resources, but also hardware with corresponding computing power, depending on the tool. This is how the motivation arose to investigate the software quality stack, how the performance looks like on a system with limited computing power. The question arose whether the software quality stack could be created as a cluster on several small computers.

But what is the motivation? Well, the individual tools of the software quality stacks are mostly installed on different servers, but these are only virtual. In fact, all tools are hardware-technically on one server and by using the same resources. If someone wants to use a separate server with its own hardware for each tool, this would be associated with significantly higher costs. Smaller companies in particular cannot afford such high purchases. In addition to the hardware costs, there are also energy costs, especially if you run many servers at the same time without being constantly productive. For these reasons, the idea was born to check whether a software quality stack can be realised as a cluster on several small computers. The main feature of our architecture is that we use a Raspberry Pi for each tool used, which is finally combined into a complete software pipeline. In this paper we show that it's possible to install a complete software pipeline on different networks with multiple instances of a Raspberry Pi 3.0.

This paper is divided into six chapters including this introduction. The second chapter explains the current state of the art. Based on this, the next chapter describes our software quality stack and its special features. The conclusive results are presented in the chapter "Evaluation", which will be discussed below. In the last chapter possible future enhancements of this work are shown.

2 Related Work

"There is a relation between quality of software products and quality of the process used to build them. Implementation of a process aims to reduce rework, delivery time and increase product quality, productivity, traceability, predictability and accuracy of estimates [1]". As shown in Figure 2, the software quality stack consists in most cases of the components "Code & Commit", "Build & Config", "Scan & Test" and "Deploy".

Fig. 1. Selfmade Example to show the components of a software quality stack.

Code & Commit (Repository) – Software projects are getting bigger and bigger and last but not least, they are no longer programmed and supervised by individual software developers, but by a multitude of them. Each one of them has different tasks and areas which he has to work on and to maintain. The repository (code & commit) was introduced in order to be able to track changes made by others, to be able to work on the same project at different locations and always be up to date, but above all to merge smaller program parts together to form a complete software package. What is also known as source code management or version control is today covered by a variety of tools, such as Git, GitLab, GitHub, Visual Studio, eclipse to name a few, and is a central component of software development.

Build & Config (continuous integration) – If changes are made to the source code, they must also be published regularly, or in other words, they must be integrated into the existing software. This task is called ”continuos integration” or ”continuops delivery” and is usually performed by tools such as Jenkins, Gitlab CI or Buildbot, which, as the central administration unit of the software quality stack, performs or triggers all other tasks. If the ”CI/CD-Tool” receives a new ”push” from the repository, further steps such as code analysis, build artifacts and deployment are triggered, monitored and error messages are issued.

”Jenkins serves as the primary interface between version control systems and deployment, and it also manages the container build process for each image repository. [2]”

Scan & Test (Code Quality) – ”Software architecture is the result of a design effort aimed at ensuring a certain set of quality attributes. As we show, quality requirements are commonly specified in practice but are rarely validated using automated techniques. [3]”

In order to develop and provide software with a high degree of quality, it ought to be regularly checked. Especially for larger software projects, manual checking of the source code would be time-consuming. In order to do this faster and easier, a static code analysis is performed autonomously within the software quality stack. For this purpose, the source code is subjected to a series of formal checks during which errors are detected. The resulting improvements are intended to ensure that the software functions without errors.

Deploy (continuous delivery) – ”Docker is a new but already very popular open source tool that combines many of these approaches in a user friendly implementation, including: (1) performing Linux container (LXC) based operating system (OS) level virtualization, (2) portable deployment of containers across platforms, (3) component reuse, (4) sharing, (5) archiving, and (6) versioning of container images” [4]. ”Containers are a lightweight virtualization method for running multiple isolated Linux systems under a common host operating system.” - [2] Thus, the Docker enables the virtualization of specific software modules necessary for the project and makes them available for use in small containers. Docker is useful during development and deployment and comes with applications that use modules without the need to install software on different platforms. It makes it easy for all developers to use the same software versions while working on the same project and prevents dependency hell, rotting code, and versioning problems.

3 Our Software Pipeline

3.1 Structure

What a basic software quality stack looks like has already been described in section 2 Building on this, we developed our own software quality stack and chose the tools Gitlab (Repository), Jenkins (CI/CD), Sonarqube (Code Quality), Docker (Build & Config) and Docker-Compose (Deployment). These should form our basis in order to create a fast and comfortable platform for the development and supply of a software project. The use and networking of these tools are described below.

In the first part of our own software quality stack is the repository to make the code we have written available for the other tools. On the first Raspberry Pi, we have set up a Gitlab server whose goal is to build a bridge where all developers can share, repair, and combine code when working in a larger group project. To give the developer, but also the other tools, access to the repository, we have set up port forwarding. The web application is to be stored in the repository for the test phase; it represents the software to be provided.

The second step is to set up the first step for the CI pipeline. Jenkins was installed as a ”Continous Integration Tool” on a Raspberry Pi and set up so that once a day it gets the current status from the Gitlab server. To further process this data, 2 plugins were installed, one for static code analysis (sonarqube) and one for image creation and publishing (docker). Once Jenkins has successfully loaded the repository, it first performs the static code analysis and then starts the build. This, as shown in Figure 3.2, makes Jenkins the ”butler” that connects all the important tools for the pipeline.

Static analysis helps with the clean code part of the project. It examines code and sets rules for code design standards that all developers should follow for the project. The aim is to identify and avoid possible errors that may occur during the development phase. This has the advantage of increasing code quality, making it cleaner and more comprehensive.

Artifactory was developed to support the integration of all required artifacts and packages for a project. In our software quality stack, this step is performed during the Build & Config stage inside a Docker Container. In this process, Artifactory passes over the necessary Artifacts to the Docker Container, which can then be cooperated into the Docker Image. If all the required data is present, including the Webserver, a new ”build” is started by Docker-Compose at the end. The ”Build” creates an image file, which goes back to Jenkins and from there to the test or production environment.

3.2 Special distinctions on microcomputers

Important differences in the use of the tool JFrog Artifactory on the Raspberry were due to the low performance of these computers. Artifactory has a very high memory requirement of 2 GB due to the used Java Heap. Unfortunately, the Raspberry with its 1 GB of RAM provides too less and also needs resources for the operating system. Increasing the swapfile to 6 GB did not result in the desired success. After a short runtime without specific configuration and any stress of the tool, the load on the memory and a little later also the CPU load increased to 100%, which caused Artifactory constantly to crash after a little while and to restart. Because of we just needed few packages and could manage and deploy our artifacts directly with shell scripts we didn't need Artifactory at all.

At the time of building the pipeline, the problem was that SonarQube did not officially support the Raspberry. Unofficially there were workarounds around this problem. However, the Java wrapper had to be downloaded as a single application and had to be separately compiled for the processor architecture (ARMv7 or armhf as a modern 32-bit version of the Raspberry with hardware floating point support). Another special feature was to replace the Java wrapper within SonarQube, as there was no start script for ARM or the Raspberry.

The installation of Jenkins itself arranged relatively easily on the Raspberry, the maintenance and use itself, however, is very laborious because the surface reacts relatively sluggish. Also, setting up and using plugins did not work right afterwards because of Jenkins could not establish an SSL connection to its update server. However, this could be bypassed by simply using the HTTP protocol instead of HTTPS for this connection. In the meantime there was also a problem that after an update suddenly the Java path did not longer fit and therefore the Java Runtime was not found. After adjusting the path, Jenkins ran again. Also the updates has to be maintained manually and not over the web interface.

4 Evaluation

4.1 WebApplication

The software quality stack we created will now be tested using a small web application and evaluated for performance and reliability. The web application was implemented in

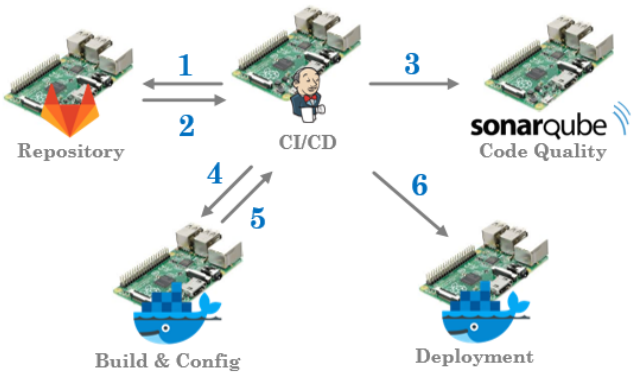

Fig. 2. Selfmade Example to demonstrate how our Softwarepipeline works.

NodeJS/Angular and consists of a graphical user interface which displays the weather data of the current, and the next four days.

Basis of this example application is a MongoDB [5] in which all the necessary weather data is stored, which is being updated once a day via a weather API [6]. Thus, all required weather data weather data of today and the next four days can be made available to the web application on a daily basis. Before these are indicated on the surface, a computation of the daily average, minimum and maximum temperature takes place. All weather data is then displayed in the form of an image, as a pure numerical value and in the form of a diagram. Within the web application you can choose between different cities. In order to adapt the example application a little to the actual web applications, some unit tests were added. This weather page is used to test and evaluate the developed and implemented software quality stack.

4.2 Practical Implementation & Configuration

In order to answer the question ”Can the software quality stack be mapped as a cluster on several small computers?”, two different tests were carried out to examine two essential aspects. The first is to evaluate whether the installed software quality stack works reliably, so that it can be guaranteed that all software changes or enhancements are published cleanly and regularly without the error occurring.

For this first test, the ”build trigger” in Jenkins is set to 15 minutes. This causes the current status of the Gitlab server to be loaded every quarter of an hour, the build and static code analysis to be started, and finally the image with all necessary artifacts to be published in the docker. The duration of the test was 8 hours, which corresponds to 32 builds. The purpose of these builds is to verify whether the software quality stack is working reliably or whether errors occur, so that depending on the type of error, regular publishing is not possible. The project was not modified during the testing phase. Neither improvements nor extensions have been made. We not only examined how often the build is successful, but also in which time the whole

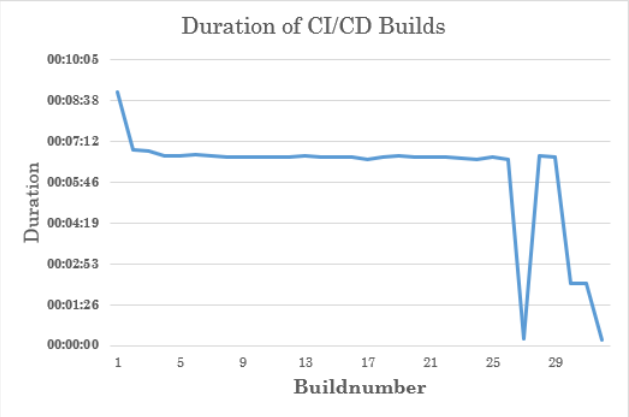

Fig. 3. Selfmade Example to show the Results of the first Test "Dependability".

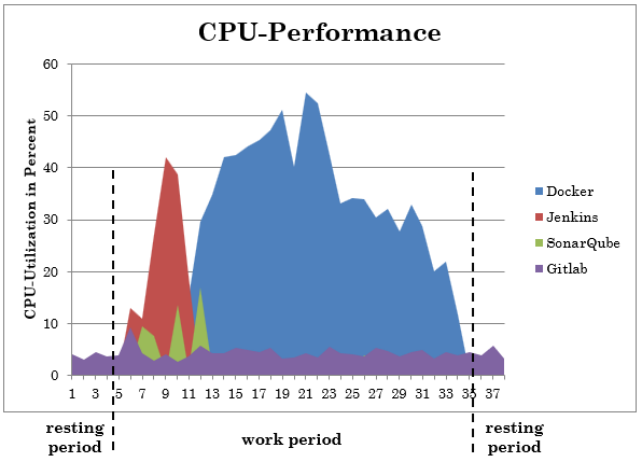

Fig. 4. Selfmade Example to show the Results of the first Part of the second Test "CPU-Utilization".

"CI/CD process" is completed. The results of the "Reliability" test are described in more detail in the Results section.

Furthermore it was tested how the software quality stack behaves in terms of performance when the entire "CI/CD process" is triggered periodically. Since Raspberry Pi's only have limited computing power available, it is of interest to know whether or not this is sufficient. Therefore, we record the CPU and memory usage of each Raspberry Pi in the cluster in 4 consecutive builds. If the CPU or memory usage is too high, this could be an indication that the requirements of the tools are greater than the available computing power. In order to be able to make a sufficiently precise statement in this respect, the memory and CPU utilization are written to a text file every 10 seconds in this test. As in the first test, the build is started every 15 minutes. If the data is logged, a statement can be made regarding the rest phase and the working phase.

With the help of the logged timestamps, we were able to merge the builds to a general build. For this purpose, the mean value of the 4 values of the individual builds at each time of the build was formed, whereby deviations were relativized and the course of the newly received build was given a general validity. We did the same with memory.

4.3 Results

Basically, the results of the first test indicate a reliable software quality stack. With a few exceptions, the duration of the builds is stable and largely reliable, as long as no connection interruptions occur. As shown in Figure 4.2, only 4 of the 32 builds performed deviate from the otherwise almost constant duration. These aborts can be traced back to a faulty connection to the Gitlab server, so that all further steps could not be carried out at all. Since the Raspberry Pi's are connected to each other via different home networks at different Internet providers, the software quality stack can only work as reliably as the connections of the Raspberry Pi's to each other are upright. Why the Gitlab server was temporarily unavailable could not be determined, since neither an Internet failure nor provider problems occurred. Otherwise it can be said that if the connection is correct, the software quality stack works for small software projects.

Now we look at the results of the load test. As described in section 3, we have created a generally valid build process with regard to CPU and memory utilization. The rest and work phases of the individual tools (Raspberry Pi's) can be seen very well, as shown in Figure 4.3. As soon as the build starts, the CPU load of the Pi's for the Gitlab server goes up for a short time, but at no time is it higher than that. Simultaneously with the Gitlab load, the CPU load of Pi's for Jenkins increases, which is higher than that of the Gitlab server, but only becomes significant when Sonarqube is started in the second step. This Gitlab server load matches the current repository load that Jenkins needs for all further steps. Once loading is complete, static code analysis will begin. This can be clearly seen in the diagram by the 3 green peaks. The CPU usage of Pi's for Jenkins only drops again when the Docker comes into play. This causes a longer and higher load than the others, but with a peak value of about 55% it is not in the high range. In general it can be stated that the CPU utilization of Pi's is in a good range during the work phases. Short-term peaks also fall off again and none of the Pi's is almost in danger of being overloaded.

In contrast to the CPU load, the course of the working memory load is different. The Gitlab server, as well as Sonarqube provide as shown in figure 4.3 for a permanently high utilization (rest phase and work phase), which remains however also during the build process very constant and contains a reserve of 10%. During the rest phase, Jenkins and Docker also have a constant workload, but at 62% (Jenkins) and 19% (Docker) they are far below the danger zone. In both cases, it is possible to see when the tools are doing their job, as both increase during the build process. The load on the docker in particular increases sharply, so that it more than triples. Similar to the CPU load, the docker is active for the longest time and reaches its peak after about

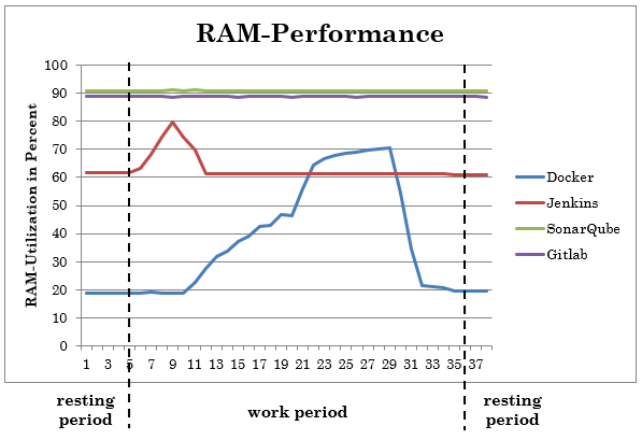

Fig. 5. Selfmade Example to show the Results of the second Part of the second Test "RAM-Utilization".

half the time.

Above all, Gitlab servers and Sonarqube ensure a high memory utilization, which is why it cannot be said exactly whether the performance is sufficient. Using our web application, the entire CI/CD process runs cleanly, reliably and has sufficiently good performance. Therefore, our created software quality stack has been successfully implemented, but needs to be explored and extended as described in the next sections. Only then can a quantitative and generally valid statement be made.

5 Conclusions

The available results are in a good range for this small example application, but should be investigated further, because in the end the output does not yet represent the complete result. Especially when larger projects are used, further tests would have to be carried out and evaluated with this software quality stack. Basically, however, it can be said that the goal was achieved, since the described software quality stack can be used to automatically publish small software projects.

Difficult and not always fully controllable is the network structure, which cannot always guarantee a continuous connection, because network problems at the provider, this cluster of small computers pose further problems.

6 Future Work

In this paper we showed that a software quality stack can be mapped as a cluster on small computers. The special thing is that we used different Raspberry Pi's, which were located in different home networks. For the evaluation we used a small self-written web application called "weather data".

The tests carried out were to be extended and quantified. Thus one could carry out a test with larger projects in order to obtain further information about the performance. A further assessment of the duration would also be possible.

It remains to be seen whether the software quality stack used will run reliably and in an acceptable time even for larger projects. This could be checked in another paper, because the question arises: "Is the computing power of the Raspberry Pi's also sufficient for larger projects? Furthermore, changes in the software should take place during the test phase in order to be able to assess the further performance of the software quality stack more precisely. Furthermore, a complete use of a tool for the creation of the "build artifacts" remains.

In future projects, for NPM package repository, it would like to use the default npm repository that is provided by JFrog Artifactory Pro. So we'll probably be able to consider creating a repository per project, because JFrog's Artifactory can be considered as a universal binary repository manager, like this we can manage multiple applications, their dependencies, and versions in one place. In addition, it is possible to standardize a package type across all applications, no matter the code base or artifact type.

References

[1] Gomede, E., and Barros, R. M., 2013. "A practical approach to software continuous delivery". p. 4.
[2] Canon, R., and Jacobsen, D., 2018. "Contain this, unleashing docker for hpc". p. 8.
[3] Caracciolo, A., Filip Lungu, M., and Nierstrasz, O., 2014. "How do software architects specify and validate quality requirements?".
[4] Boettiger, C., 2014. "An introduction to docker for reproducible research, with examples from the r environment". *arXiv preprint arXiv:1410.0846*, p. 9.
[5] MongoDB, 2018. Getmongodb.
[6] OpenWeather, 2018. Openweathermap.