IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 21, Issue 6, Ser. I (Nov - Dec 2019), PP 46-55
www.iosrjournals.org

Avoiding frequently failing tests in Continuous Integration

Denislav Lefterov[1], Svetoslav Enkov[2]
[1](Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)
[2](Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)

Abstract: *By introducing Continuous Integration and deploy automated tests across multiple instances, runs and builds, the risk of accidental and frequent test failures increases instantly. These failures may not relate to a specific bug, problem or issue, they can be due to many other external factors, such as miss-configuration in the CI platform, performance issues, configuration of the project, lack of hardware resources or incompatibility between resources. In order to avoid these types of behavior, there are several factors that can be considered and need to be taken into account. In this article, we will look in detail at the measures taken on frequently failed tests, related to bad configuration. This paper provides a mechanism for how we can avoid the frequent failures of automated tests. What are the good practices that we need to follow in order to be able to develop a stable automated process oriented towards continuous integration?*

Keywords: *quality, assurance, testing, tests, frequent, failing, automation, continuous, integration.*

Date of Submission: 22-11-2019 Date of Acceptance: 06-12-2019

I. Introduction

Recently, software development of Web and Mobile applications have become an integral part of our daily lives, every action in our lifestyle is invariably linked to the surrounding Cyberworld. Last years the development of such types of technologies has grown and gained momentum on a large scale. This invariably entails an increased risk of problems, issues or bugs in the systems. In order to avoid and reduce such behavior, the software industry has increasingly necessitated the use of Continuous Integration (CI) systems in complex software development and the successful launch of new products or the improvement of existing ones.

Continuous integration is expected to provide several benefits such as: getting more and faster feedback from software development process, developers by itself or even clients. Frequent and reliable versions that lead to improved customer satisfaction and product quality. On the other hand, continuous delivery between development process and development teams is enhanced and manual tasks can be simplified and minimized. The increasing number of manufacture cases indicates that continuous integration processes are entering industrial software development experience in different areas and sizes. In the same time, adopting continuous practices cannot be considered as a trivial task of organizational processes, practices and tools may not be ready to support the highly complex and challenging nature of these habits. Because of that complex and multi-layer solutions must be applied at all levels of software development life cycle (SDLC).

II. Problem area

With the implementation of continuous integration tools we ensure better way to deploy automated tests across multiple instances, runs and builds, the risk of accidental and frequent test failure increases instantly. These failures may not relate to a specific bug, problem or issue, they can be due to many other external factors, such as miss-configuration in the CI platform, performance issues, configuration of the project, lack of hardware resources or incompatibility between instances and standards. In order to avoid these types of behavior, there are several factors that can be considered to be taken into account, further explained in this case study. In this article, we will look in detail at the measures taken on frequently failed tests, related to bad configuration.

The application that was conducted on the survey is Java-based; it uses two different types of databases: MySQL and PostgreSQL. And the well-known multi-layer architecture for development based on many types of services as a Back End core. This project is oriented towards health insurance purposes, processes a large number of user requests, invoices, payment orders, and big complex business logic. Front End uses Angular JS as the basic interpretation for graphical units. This study aims to demonstrate an approach how Web applications are being developed using this similar route and also the smooth transition to continuous software test automation in the Scrum process. We will follow important steps to ensure a smooth transition to continuous integration and will demonstrate how to avoid common test failures and a better approach to strengthen our Pipeline build. Finally, we will summarize how all these practices have affected our project and what is the outcome of this research and key results.

III. Test Automation

Test automation increases the depth and speed of the quality assurance process; it is the core of Agile and a key element of successful development. Test automation uses software to monitor test performance, compare actual results with predicted results, set up test preconditions, and other test measurement features.

The software framework is an application programming interface that is used to facilitate the automation of test cases. Examples of frameworks include JUnit on Java, "Robot Test Framework", SeleniumHQ, and Selenium Web Driver. The architecture is integration of multiple frameworks into one integrated system. From application engineering point of view, one framework is similar to an individual independent component and architecture is a system of fragments where at least two are integrated to achieve new capabilities.

IV. Continuous integration

Continuous Integration is a practice where small changes are often tracked. Any changes to the code are noted, packaged, and tested, and its main purpose is to prevent problems with the integration of new versions. The rapid adoption of the Agile methodology on such a large scale increases usability. Most servers with continuous market integration nowadays have a basic application programming interface to extend the functionality. This has led to the development of a large number of open source plugins, which increases their flexibility and utility. The server performs tests at several levels (unit, functional, non-functional and integration) and includes implementation of new operations in individual aspects. This provides multiple metrics at each stage to track the processes and progress of an application [1].

1) Continuous Integration validates the code that was pushed to a given branch of repository

• Static code analysis: Reporting the results of static code execution;
• Compile: Generating the executable files by linking the code and compiling after;
• Unit test: Writing unit tests, executing them, checking code coverage and reporting the results;
• Deploy: Build the code and install it into a test/stage/production environment;
• Integration test: Providing results by executing the integration tests;
• Automation functional test: Providing results by executing automation related tests;
• Report (dashboard): Indicating the status of key parameters by posting Red, Green, and Yellow colors to a publicly visible location, such as version control systems, test management systems and etc.

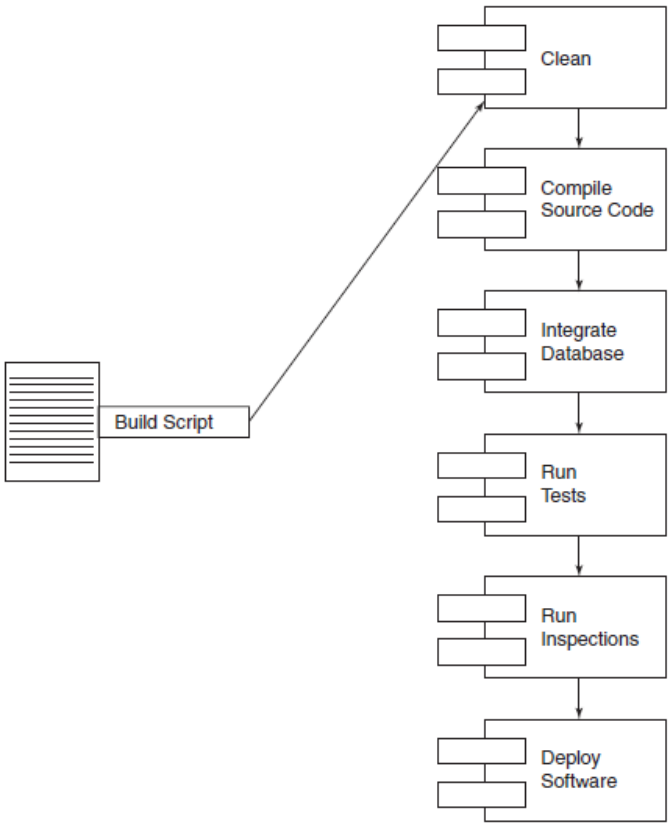

Figure 1: Main common phases, which one build script includes in Continuous Integration [1]

2) Cycle of Continuous Integration includes

Continuous Integration helps testers to perform automation effectively to uncover defects at a faster rate and improve the regression results. After execution this will give the percentage of test coverage area and detailed statistics. These values will be transferred into big scaled report which will cover user stories and functionality mapped to the product increment. There are several tools that are being used by organizations as a part of continuous integration to build the automation scripts. Few examples of those tools are JUnit, Selenium, SONAR Cube, etc. [2]

3) Advantages of Continuous Integration

• Enables a quick feedback mechanism on the build results;
• Helps collaboration between various product teams within the same organization;
• Decreases the risk of regression since the code is tracked;
• Maintains version control within for various product and patch releases;
• Allows earlier detection and prevention of defects;
• Provides a facility of falling back to previous versions in case of any problem to the current build.

4) Risks and challenges of Continuous Integration

• Tools maintenance and their tech administration have associated with costs to it.
• Continuous Integration guidelines need to be well established before starting it.
• Test automation is a rare skill in the market yet and existing testers may have to be trained on that.
• Full fledge test coverage is required to see the benefits to automation; the team should be very patient.
• Teams sometimes rely too much on the unit testing and ignore automation and acceptance testing; this could lead to working code, but misunderstood business requirements.

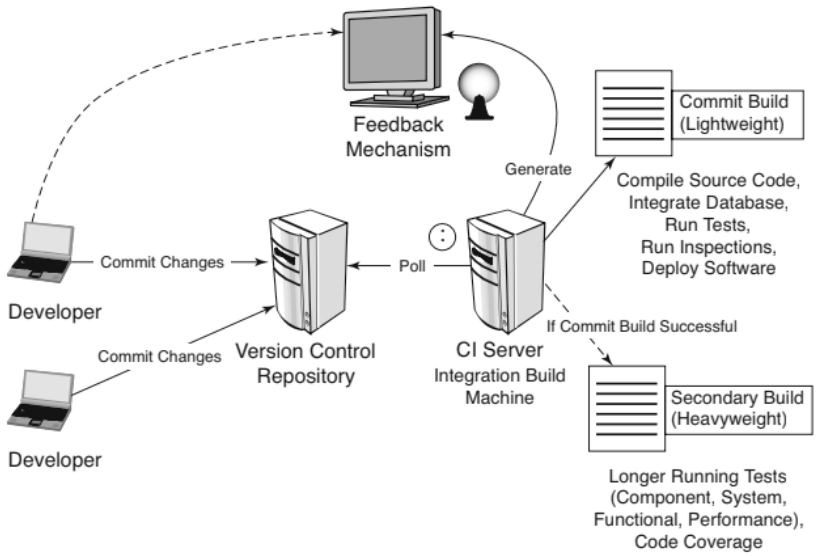

Figure 2: Topology how code is tracked by continuous Integration and Version Control Systems [1]

5) Different types of tools supporting CI are:

There are many tools, which purpose is to build continuous integration and continuous delivery mechanisms, popular tools are: Buddy, Team City, Jenkins, Travis CI, Bamboo, GitLab CI, CircleCI, Codeship and many more. Despite the well-known business advantages of test automation, it is important to understand that automation is not a magic wand. There are several factors that can compromise the integrity of a test automation solution. For example, the automation engineers are responsible for deciding whether an organization should accept test automation; they often overlook the factors that can influence success. [7]

6) The most common reasons for test automation failure are:

• Lack of experience in designing and/or implementing frameworks. Inability to incorporate improvements;
• Lack of proof of concept (POC) on pilot phases;
• Lack of in-depth analysis of test reports or stack trace reports;
• Lack of clarity on what to test and what to exclude in tests;
• Lack of exceptions and functionality logs for debugging/stack trace;

• Improper maintenance of site storage/ security vulnerability/GDPR concepts. [6]

In addition, the automation solution is complex and fraught with several disputes, such as:
1. Will one organization need an end-to-end automation testing approach?
2. What is the optimal measure of automation? How could be achieved?
3. Are test cases required for all features in the application, large or small amount or epics oriented strategy for planning, when writing stories/tasks?

There is no one-size-fits-all answer to these questions as they depend on subjective factors such as teams, projects and organization. The best solution is one that meets the unique requirements of an organization and drives collaboration between development, QA and business teams. This can guarantee transparent and effective communication within the time-frame and provide the necessary scope and quality of practical experience for the end user. Lack of effective communication between these three teams can lead to critical failures in the process. Thus, companies must first ensure that they have identified the "right reasons" for introducing test automation. Once organizations have identified the needs for automation, they face a number of challenges when it comes to implementation, most of them are. [2] [6]

• Poorly designed software architecture — as a building requires a solid foundation, software must be designed for a robust and sound architecture. The inability to integrate key scenarios, design common issues, create risk mitigation and factor in the implications of long-term key decisions can jeopardize the integrity of any Web application. While companies can use modern tools and platforms to simplify the task of building applications, there is a need to continue with the utmost precision and care in designing applications to ensure that all scenarios and requirements are covered. The lack of such careful monitoring can lead to poor architecture, leading to unstable software, unable to support existing or future business requirements and challenging to deploy or manage in a production environment. Consider the following example: A leading software vendor initiated an automation solution without a clear vision, resulting in a poorly designed framework. This led to several complications during the implementation phase as the framework failed to support multiple objects / streams and made scripting difficult. This has led to considerable time and cost spent fixing the frame rather than focusing on results. [2] [6]

• Incorrect identification of automation goals — when deciding the type of automation framework to be implemented, it becomes critical to first identify the organization's automation goals. Framework work must be selected on the basis of its capacity to achieve these objectives. Companies that do not align their goals with the functions of the framework run the risk of incomplete automation coverage and increased costs.

• Choosing the wrong automation framework — Data-driven and Key-word frameworks are two of the most commonly used frameworks. Each has its advantages and disadvantages. Companies need to be aware of the differences between such automation frameworks in order to choose the most appropriate one to fulfill their organizational goals.

When planning to choose an automation framework, companies need to make sure that the architecture approach provides facility and tool support, and also customization. Improper valuation and basis can lead to a negative return on investment and big delays in release basis. [2] [6]

V. Integration in Jenkins

In this study, we will review a narrower approach through an automated continuous integration tool Jenkins. This is a free and open source automation server written in Java programming language. Jenkins helps to automate the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat. It supports version control tools, including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, TD/OMS, ClearCase and RTC, and can execute Apache Ant, Apache Maven and based projects as well as arbitrary shell scripts and Windows batch commands. The creator of Jenkins is Kohsuke Kawaguchi.[4] Released under the MIT License, Jenkins is free software.

Builds can be triggered by different ways, for example by commit in a version control system, by scheduling, via a cronjob mechanism, by requesting specific build URL parameters. It can also be triggered after other builds in the queue have completed. Jenkins functionality can be extended with many developed plugins.

It is very often that for example in integration tests, perhaps some batch tasks running in parallel or with a background thread may also violate performance value of testing. If the script is run on a real environment, are there any other external requests that might affect the pipeline build? Are there other tests running at the same time, in parallel? In addition, in asynchronous applications, the execution order may sometimes not be taken for granted or by priority. Now we will follow up on the important cases where we may have similar problems with frequent test failures due to missed settings steps in the pipeline build. [3] [8]

• **Caching**: Caches must be taken into consideration when designing tests. Sometimes due to time manipulation (time laps), caches or stagnant data, test results can become unpredictable. Specs could frequently fail, because of old content was loaded or the new content was not loaded properly. In some other cases after the code was executed once, this could cache a lot of the dynamic functionality and mislead the performance on our app. It is very important to measure the first execution which was made, and we should consider clearing the main cache in every single build if there is caching stored after deploying the application.

• **Cleaning status step**: A good test should always adjust its expected point environment and cleanse any personalized behavior to a vanilla state. This is one of the most difficult unpredictable outcome scripts to identify as it is not one that fails, but all tests may be affected. Most IT organizations have this issue. Clearing step is important for further independent testing. Build after build or different test areas in execution.

• **Dynamic Content:** When testing user interfaces, we want tests to perform as expected. Sometimes the script may have to wait for the dynamic content to load first, for example FE. There can be asynchronous data calls which can be delayed, due to infrastructure slowness or milliseconds delay. In this situation, explicit specific wait should be obtained or implicit internal wait should be created in our automation framework. In this way, we ensure that the tests will not fail due to delay in dynamic content. [4]

• **Temporary "Timed Bombs"**: What time zone is used and what is the time zone of the application or the server? We should not make the assumption that test will always run in the same time zone as it was designed. We should accurately measure time intervals. For example a single test collects invoices of the current date. Number of invoices will change frequently. We have to follow the same logic as the test class. If test is limited by the current time or by the whole day (24 hrs.) we should consider all special cases with timing.

• **Infrastructure Issues:** In some cases, the reason for frequent fail is not because the test is flake. It may be test framework error, selenium driver error, other library, or version changed also problem with this version of the browser and etc. this can waste a lot of time trying to figure out what's wrong with the tests. Other incidents, such as failures at a continuous integration node (CI), network problems, database disruption, misconfigured module and many more can be the reason for infrastructure issues. We should always check our used technologies and their collaboration with the infrastructure.

• **Third-Party Systems:** Integration tests that are not working in a difficult external environment inevitably depend on third-party systems. Good practice is also to check the correctness of external systems, if available, and every component that systems interact with. There should be tests that confirm integration with external systems. A good strategy is to export these specific tests to another package. Also, a good approach will be to try deleting all external systems when checking the integrity of the system. These tests are called Integration specs related to 3rd party system tests. [3]

7) Retry on Failure

One main solution to the issue is the configuration setting integrated in most CI automation tools, called retry on failure, in this way the failed area will be retried n times as we wish. In this way, we insure that our one-off test will not ruin the entire build.

Some example code configuration in CI (for example Jenkins) set tool can be by triggering nodes:

```
node{
   def job_list = ['job 1', 'job 2', 'job 3'] //Jobs in CI build configuration
   for (job in job_list) {
      stage (job){ // Testing environment
         println ("${job}")
         println("default: currentBuild.result: ${currentBuild.result}")
         try{
            retry(3) { // There will be 3 retries if a test fails
               println("execute ${job}")
               if (job == 'job 1') {
                  throw new Exception() // Exception handling
               }
               currentBuild.result = "SUCCESS"
               println("Job was successful. currentBuild.result: ${currentBuild.result}")
            }
         } catch (e) {
            currentBuild.result = "FAILURE" // Exception handling
            println("catch exeption. currentBuild.result: ${currentBuild.result}")
         }
         println("result: currentBuild.result: ${currentBuild.result}") // Print results in the end
      }
   }
```

}

In some cases, this approach is not recommended if cache is enabled, because after the first run, there can be cache storage and if we have a performance issue, then it can be masked in this way after the second attempt to pass the test.

8) Optimize builds process, if they are slow

Large code-bases can cause the integration of software components to take considerable time. To determine if the problem is related to the size or integration of these components, check how long the compilation step takes. If this step turns out to be time consuming, perform a gradual build instead of a complete build.

The incremental build will only compile and regenerate the modified files. This can be risky because depending on how this is done, it may not get all the benefits of CI, and build could break. An effective CI system is associated with risk mitigation and ideally the integration environment should be cleaned by removing old files and then compiling / regenerating the code to effectively determine if something is broken. Therefore, use incremental builds only if needed and explore other areas that lead to slow builds. Some zones may be scaled up. For example, if there is a Java system with a natural DLL or a shared object library that rarely changes, it might be wise to restore this library only once a day or in a single iteration. In fact, some may argue that this rare DLL or shared object is treated as a separate CI project and treated as part of the project using dependencies.

9) Building System components separately

In most cases, building integration takes a long time to execute because of the time it takes to compile the source code and other dependable files. In this situation, this can divide the software into smaller subsystems (modules) and build each of the subsystems individually. To build application components separately we create distinct projects for each subsystem undependable one from another. This can be done by the CI system - just make one of the main projects as subsystems. If there are changes to one component based on dependencies, the other projects are also restructured and will speed the process of building code.

It's almost mandatory to avoid brittle dependencies in our CI platform, every dependence leads to bad unstable build and pre-configuration will be complicated. In the next figure, we will illustrate 3 main steps how we can steady our CI integration build and that will guarantee smooth implementation.

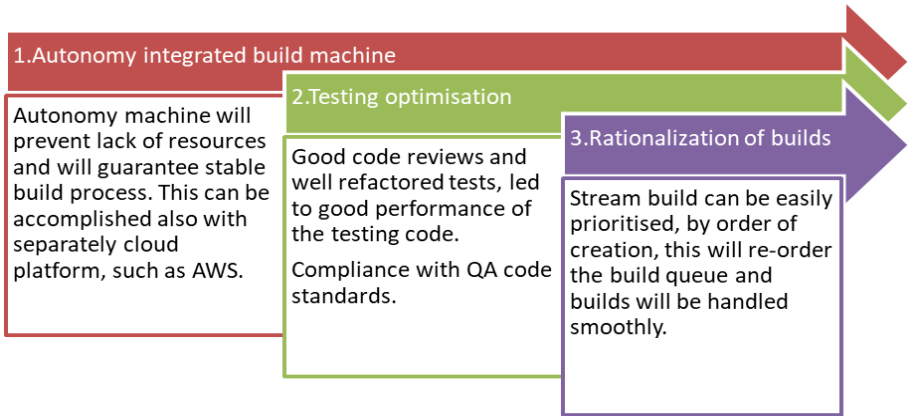

Figure 3: 3 – Steps to guarantee smooth and hassle-free build to continuous integration success

VI. Case Study to our project

As we said in the beginning our study is focused on a project using the following structure and technologies: Java Web-based application which uses different types of databases: MySQL and PostgreSQL. Project was separated with multi-layer architecture for development based on many types of services as a backend core transmitter. The application is oriented towards health insurance purposes, processes a large amount of user requests, invoices, payment orders and big complex business logic. Front end uses Angular JS as the basis interpretation for graphical units. In this complex application, we collaborate between different portals. They are very dependent on each other, for example: Employer, Member, Invoice and Search.

In the very beginning of the project, we were not facing any difficulties; all automated tests were passing as expected. New UI and API tests were developed and project was growing on a large scale. As we mention early our project is based on microservices architecture and a lot of new services and gateways were introduced while the application grew.

Our study was conducted because after 14 sprints and adding a large number of automated specs, we started to experience problems and issues related to frequently failing tests or in other words flaky tests.

We were searching and investigating these problems in our tests, but we were following them locally, they passed in 100% of the cases they have been run. In general, the UI automated tests were the main problem because CI builds were configured to run tests in parallel, which means that every single test related to FE part will launch Web Driver for browser control. This took all the resources and delayed the work of the compiled code (our application) in the build phase. Most of the stack trace errors were timeouts for not found/missing elements or cached old content was loaded, even some of the tests were not executed at all and the whole build was interrupt from error interception from the server.

Most of the exceptions from testing framework and Jenkins were:

- *WaitTimeoutException: condition did not pass in 30 seconds. Failed with exception: Assertion failed;*
- *The following element was not found/missing:'button, element, view';*
- *Services were not steady / running;*
- *Failed database migration script;*
- *Failed flyway implementation script;*
- *Back end API failed with response status code 500;*
- *Gateways and Services did not respond correctly when tunneling between each other.*

As we said in the good practices for CI configuration, there is a better approach to handle failed tests, and this is the retry on failure closure. In our example, we extended this setting to retry the failed test N times and after all retries the test will fail and will log a proper stack trace error, in this way we will ensure what is the actual cause of failure. Here is the class example of our automation framework extended from the given example retry on failure block.

```
class RetryTest {
        //define retry implementation code here
  public class Retry implements TestRule {
    private int retryCount;

    public Retry(int retryCount) {
      this.retryCount = retryCount;
    }

    public Statement apply(Statement base, Description description) {
      return statement(base, description);
    }

    private Statement statement(final Statement base, final Description description) {
      return new Statement() {
        public void evaluate() throws Throwable {
          Throwable caughtThrowable = null;

          // implement retry logic here
          for (int i = 0; i < retryCount; i++) {
            try {
              base.evaluate();
              return;
            } catch (Throwable t) {
              caughtThrowable = t;
              System.err.println(description.getDisplayName() + ": run " + (i + 1) + " failed");
            }
          }
          System.err.println(description.getDisplayName() + ": giving up after " + retryCount + " failures");
          throw caughtThrowable;
        }
      };
    }
  }
  public Retry retry = new Retry(3);
}
```

This retry block saves us a lot of failed builds, because in 35% of the builds there were failed tests related to UI, because our scripts are running in parallel to optimize time in some cases performance of the build was not steady, retry on failure was the guarantee, that we will ensure successful build, if the test spec failed 3 times in a row, this definitely is a caught bug in the application. This is the correct behavior of our pipeline now and after the changes, if a build fails with UI tests, now we are sure that this is 99% caught defect/issue.

By having the following tests structure via Jenkins, our tests have been separated from the following annotations. Front end and Back end unit tests are handled first, next tests are automation functional related to API and UI through different platforms. In this way, every annotation will be handled in different group and there will be no interference between portals and tests.

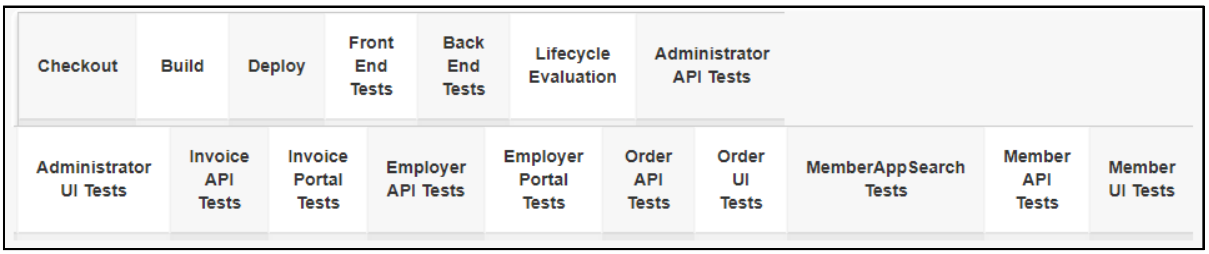

Figure 4: Pipeline state order in our current project

Checkout phase: will get the new code changes from latest commits and will prepare for build phase.
Build phase: will handle and build new code with the excising ones.
Deploy phase: will compile the whole project to be runnable and available.

First tests that will be executed are Unit and Integration oriented specs. These tests are quick and they are testing only the units of the code, in our case, they are situated in the following structure: **Front end** oriented and **Back end** oriented tests.

Lifecycle evaluation will stabilize the application and prepare the current build to handle API and UI functional automation related tests. This means that the database will be load with data dumps, which we are pointed in our Jenkins configuration. In this way by different jobs we will ensure that our application is stable and will execute all tests as expected.

A good approach that we used was cleaning the old cache content, because in a lot of cases we were loading the old content from UI perspective and cleaning the old cached storage was necessary, to obtain and check new Front end elements properly. In Figure 5 we listed the main flow how one Front end test should behave, first there should wait implicitly for element (about 30s by default) after that the cached content should be flushed from Jenkins configuration and after these steps, we should ensure back end and front end corresponds with fresh initialization. [4] [5]

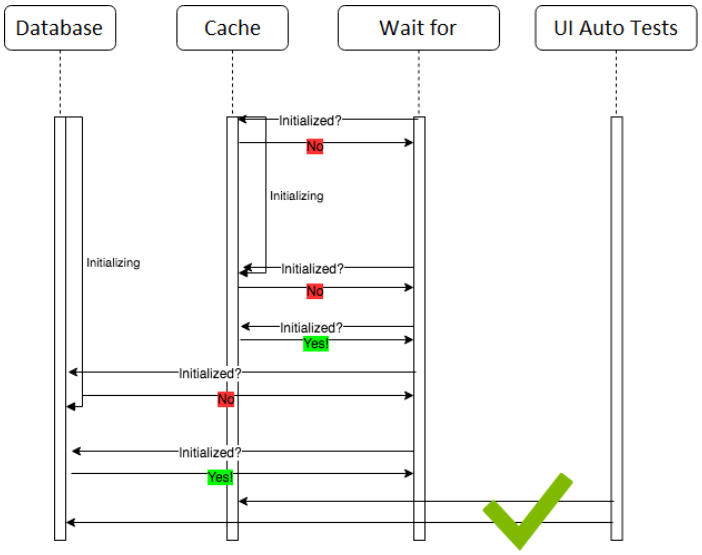

Figure 5: UML Sequence diagram showing how the flow between Caches and UI elements is handled

As we point out, when UI is extended also automated tests are increased at the same time, this led to running out of resources of the virtual machine, which is set to run CI builds with limited resource stack. There was a need to extend those resources to gain smooth builds across all branches. There was the need for us to migrate into AWS (Amazon Web Service) cloud server, for optimized and clean builds.

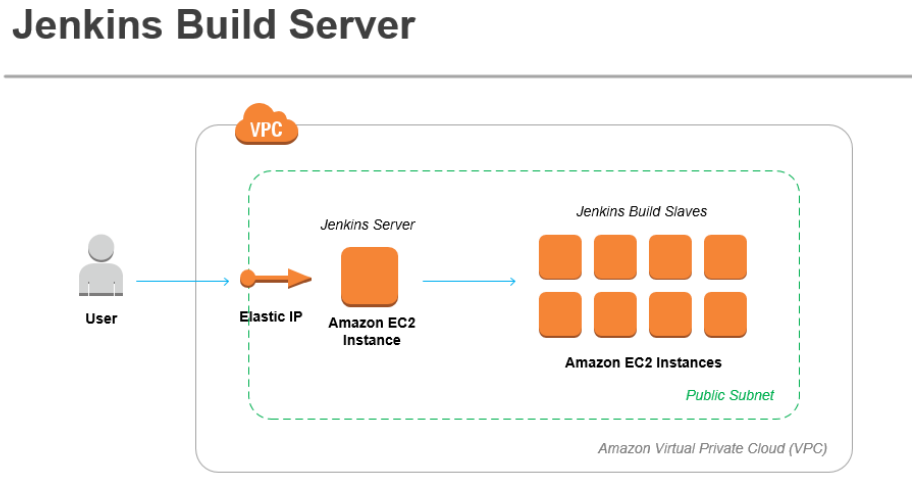

Figure 6: Amazon Web Services cloud with Jenkins integration [9]

In this figure we show, how AWS can be integrated to handle build resources of Jenkins. In this way we ensure that builds will be clean and smooth and no longer lack of resources can occur, because every time our build needs to up resources, this will be handled by the cloud virtualization and will give us the needed resources. The setup is very easy, because in AWS cloud server there is a setup wizard, who supports you how Jenkins can be migrated and if something is not clear, they have a good support to ask and help you. [8]

After all the above actions were taken, our builds stabilized significantly, we were able to eliminate the problem tests and builds by fixing many configuration vulnerabilities. The continuous integration process is now much faster and more convenient. This resulted in a 25% increase in productivity for our product and significantly exceeded the planned deadlines for launching new functionalities. [7]

VII. Conclusion

With the advent of continuous integration in software development, the bugs encountered in production environments are limited, but problems with building and running tests are compounded by many dependencies and configuration issues. In this scientific article, we dug into the problems and solutions of a Web platform targeted many users and third-party companies. We have provided a series of solutions in order to facilitate the transition to continuous integration and especially Jenkins CI build tool, also we demonstrated that there is a solution to consolidate and avoid tests that are flaky and unstable. By having these steps presented, we can be sure that problems that occur in buildings via continuous integration tools can be minimized; the pipeline of different tests will run smoothly and stable. By rearranging script build in steps we ensure that tests will run in sync and sorted by type, this is a better way when it comes to test plan report of the iteration cycle. This article demonstrated everything needed to facilitate the process of continuous integration and running automated tests to separated staging and auto environments.

Acknowledgements

UNIVERSITY OF PLOVDIV PAISII HILENDARSKI, 24 TZAR ASEN, 4000 PLOVDIV
Denislav Lefterov, PhD student, Faculty of Mathematics and Informatics

References

[1]. Paul M. Duvall, Matyas, Glover. *Continuous integration: improving software quality and reducing risk*, Addison-Wesley, 2007, ISBN:978-0-321-33638-5.
[2]. Madan Mohan Reddy B, Santosh Kumar CH - *Flexible Approach to Test Automation with Measureable ROI, Tavant Technologies Inc.,2013, white paper.*
[3]. *Flaky Tests - A War that Never Ends December 4th 2017* - https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359.
[4]. Nebojša Stričević - *How to Deal With and Eliminate Flaky Tests* - https://semaphoreci.com/community/tutorials/how-to-deal-with-and-eliminate-flaky-tests.
[5]. *DevTestOps: CI/Testing: Application Level Readiness Detection to Remove Docker Initialization Race Conditions* - https://medium.com/dm03514-tech-blog/ci-testing-remove-docker-initialization-race-conditions-96caa159bd86.

[6]. *Continuous Testing in Micro Focus* - 162-000231-00, 06/2019, Micro Focus Inc,, white paper.
[7]. Sapana Kejadiwal - *Continuous Integration Using Jenkins, L&T Infotech, white paper.*
[8]. *Set Up a Jenkins Build Server* - https://aws.amazon.com/getting-started/projects/setup-jenkins-build-server.

Denislav Lefterov. " Avoiding frequently failing tests in Continuous Integration." IOSR Journal of Computer Engineering (IOSR-JCE) 21.6 (2019): 46-55.