Cloud-Based Testing of 4G Base Station Software
Abstract

Executing test automation for large-scale software-intensive embedded systems requires a lot of hardware because tests must be executed for different hardware configurations. When there is not enough hardware for all kind of configurations, other solutions are developed to fill high test coverage with less hardware. Placing simulated configurations to the cloud makes the hardware usage more effective.

The case company has developed a cloud-based testing service for embedded systems which are in this case 4G base stations. This study investigated how well the service fulfils the target team’s needs for testing. Testers have a different kind of requirements for cloud-based testing. Requirements were split into four categories from the qualitative data which was collected by interviewing testers of the target team. Requirement categories are test environments, test automation development, test execution and partly-simulated system under tests, more commonly known as SUTs.

Four tests were implemented to the cloud with Robot Framework which is a test automation tool for developing automated tests. An empirical data showed that executing cloud-based tests is not always so fast due to long waiting times of getting a test environment from the cloud. However, when test environments were received they have initialised automatically with default settings and required testing tools. That reduces testers’ workload because locally built test environments require a lot of manual work like maintaining test environments.

Beside other research questions, this study investigated fault detection capability of partly-simulated SUTs. 34 Cloud-based tests with partly-simulated SUTs were executed. 26 Tests passed and 8 tests failed. The simulator’s software caused only one failure and tested software caused other seven failures. From these statistics, we can at least say that the partly-simulated SUT can find some faults. The study also investigated a specific fault which was not found in the simulator even though it was found in the real SUT, which was a clear disadvantage.

Keywords
Test automation, cloud testing, cloud-based testing, software testing, base station, embedded system

Supervisors
Ph.D., University lecturer, Antti Siirtola
D.Sc., Professor, Mika Mäntylä
Foreword

First, I would like to thank the case company and the team in which this research was made. Also, I’d like to thank several other people who supported me in this writing process. Without yours tips this thesis would not be in this form where it is now. Especially I’d like to thank my supervisors Antti Siirtola and Mika Mäntylä for your guidance during the process.

Also, I feel this is the right time to thank my family, friends, and all those people who have supported me during my studies. There have been several people who have motivated me during these past six years. You pushed me forward, i.e. the final stage of studies. All of You deserve great praise.

Finally, I am grateful to you Arto Rukajärvi because you hired me two years ago. During the employment relationship, I got a chance to work with great people and for the first time, I gained practical experience from my own field of study.

Joni Pennala

Oulu, May 7, 2017
Appendix C. Results of All Requirements

57
Introduction

This section will explain the motivation to do the research and research questions and the research structure.

1.1 Problem Statement

Makki, Van Landuyt & Joosen (2016) declare that continuous delivery is coming more and more popular in a software development life cycle which is also a current issue in the case company of this study. There are going to be shorter cycles to deliver a new product to customers. In the future, one product must be released within three months. Because of this, one should be able to release the product reliably at any point during this time. To reach the objective that the product stays reliable all the time, software testing should be performed continuously. It can be said that the quality of a product relates to how well the product has been tested during the development process. (Kumar & Mishra, 2016; Ghinea, 2016; Mathur & Malik, 2010; Mårtensson, Ståhl & Bosch, 2016; Karmore & Mahajan, 2016)

The case company uses continuous integration and other agile methods for developing large-scale software-intensive embedded systems like 4G base station systems. Those embedded systems are combined with electronic and mechanical systems including highly complex technical products such as different types of Remote Radio Units (RRUs), Baseband Units (BBUs) and the software. This research is made in cooperation with the company’s 4G Base Station Integration and Verification unit, more commonly known as I&V unit. (Huang et al., 2012; Holappa, 2013; Roberts, 2016; Mårtensson et al., 2016)

Karmore & Mahajan (2016) state a common fact that testing is the most time-consuming and costly portion of the embedded system development process. When a software is developed continuously, continuous regression testing, more commonly known as CRT, should be performed as the target team does. Basically, executing CRT requires test automation so that same tests can be executed frequently and continuously with the newest software build, say, once in every week. Software developers develop multiple new builds in a day with many corrections and modifications which can cause some failures to other functionalities. Because of modifications, same tests need to be executed continuously. (Kumar & Mishra, 2016; Mathur & Malik, 2010; Mårtensson et al., 2016)

The field of testing is to verify that required functionalities work with all specified base station configurations. One of the main functionalities is that a cell phone can connect to a base station’s cell. If the connection breaks, a network connection in the cell phone will not work either. A different type of hardware can be used for different hardware configurations. Furthermore, there can be differences between the number of hardware in each configuration. Same software must work with all specified hardware configurations so same tests need to be executed with dozens of different configurations. (Huang et al., 2012; Patel et al., 2016; Wannstrom, 2013; Rumney, 2013; Holma & Toskala, 2009; Mårtensson et al., 2016)

In testing, there are three objects which are a tester, a test automation tool and a system under test, more commonly known as an SUT. In this context, the SUT is the specified hardware configuration including the software which is controlled by different testing tools in a computer, because embedded system testing requires a lot of different testing programs or applications (Karmore & Mahajan, 2016). Both the computer and the SUT
are a part of a single test environment. Gomez-Miguelez et al. (2016) call a test environment as a testbed but we call them more clearly test environments. (Garousi & Mäntylä, 2016; Mathur & Malik, 2010; Pinola et al., 2013; Gomez-Miguelez et al., 2016; Mårtensson et al., 2016)

The target team has already developed locally test automation with certain tools but they cannot build enough SUTs due to limitations of the hardware. That is often a problem when companies need large test environments for testing embedded systems (Karmore & Mahajan, 2016). Test automation is a crucial part when executing continuous integration testing in embedded software development. All new software builds should be tested with 100 % pass rate but that is nearly impossible when SUT is the large-scale software-intensive embedded system. Currently, the target team has only about ten different SUTs in test automation and none of those SUTs includes even a biggest possible base station configuration. (Mårtensson et al., 2016)

The case company has provided an internal cloud-based testing service which offers a pool containing various ready-made test environments with partly-simulated SUTs where the RRU hardware is simulated (Tao & Gao, 2016). The cloud service can run automated test suites which are implemented by Robot Framework. It is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD) and it utilises the keyword-driven testing approach (Robot Framework, n.d.; Robot Framework documentation, n.d.). Each tester of the company can reserve and use the cloud’s test environments and already many teams have begun to use the service but the target team has concerns towards it. Biggest concerns are connected to the reliability of the partly-simulated SUT because a simulated hardware might not act as the real hardware and then some faults might not be found before the product has been released to customers. Generally, in the literature, it has been admitted that when simulating hardware units, there are limitations regarding for example timing (Mårtensson et al., 2016).

1.2 Research Questions

The main research question for this study is the following.

**Main RQ:** How well does cloud-based testing fit for 4G base station software testing?

The aim of the study is to understand how well the whole cloud-based testing service suits the target team’s testing purposes. The service has four different parts that need to be analysed. The first part contains test environment reservations and remote connection to the test environment. The second part is a test automation tool which is used for developing and maintaining automated tests. The third part is the web application which is used for running tests when tests are already implemented. The fourth part is the fault detection capability of partly-simulated SUTs. Because all these parts influence the services' suitability, we will create following sub research questions so that all issues will be analysed the best possible way.

**Sub-RQ1:** What conclusions are there in the literature about cloud-based testing in general?

**Sub-RQ2:** What requirements do embedded system testers have for cloud-based testing?

**Sub-RQ3:** Do the test environments of the cloud fill the requirements?

**Sub-RQ4:** Does the test automation tool of the cloud fill the requirements?
**Sub-RQ5:** Do the test runs in the cloud fill the requirements?

**Sub-RQ6:** How high is the fault detection capability of the partly-simulated SUT against the real SUT?

**Sub-RQ7:** How do the results of cloud-based testing compare against information in the scientific field?

Answers for **Sub-RQ1** will be gathered from the literature that is analysed in the second chapter. The goal is to find already known benefits and problems about cloud-based testing as a part of software development processes where we can take part. In the literature review, it is also investigated which factors must be considered when deciding to automate software testing with a test automation tool. Embedded system testing will be investigated to understand what challenges embedded systems bring to software testing. The aim is to find some information gaps in the literature, to which this research can give answers.

**Sub-RQ2** will consider concerns and possibilities that testers see in different parts of the cloud-based testing. We will collect qualitative data by interviewing testers from the target team about concerns and possibilities which will be generated to requirements list. Requirements will be divided into four categories according to four different parts of the service. **Sub-RQ3**, **Sub-RQ4**, **Sub-RQ5** and **Sub-RQ6** are generated from requirements so that each of those questions are related to own category of requirements.

Answers for **Sub-RQ3**, **Sub-RQ4**, **Sub-RQ5** and **Sub-RQ6** will be collected when implementing four tests to the cloud. Observations and empirical data will be used for justifying the results against questions (Easterbrook et al., 2008; Johnson & Onwuegbuzie, 2004). **Sub-RQ6** is a big research question because a fault detection capability of a partly-simulated SUT is one of the key concerns which was stated already before this research started. It will be analysed when reproducing some already known faults which were detected with a real SUT. Also, it is interesting to see if the simulator’s software causes new faults. That kind of faults will be collected during the process of implementing tests.

**Sub-RQ7** is the question which should bring an academically valid contribution to the scientific field because it has been stated that cloud-based testing is not a common testing method so far. (Lehtiranta, Junnonen, Kärnä & Pekuri, 2016; Raulamo-Jurvanen, Kakkonen & Mäntylä, 2016)

### 1.3 Research Method

When the research objective is a real-world problem in the real organisation, the research process is a constructive research. Developing practical problem solutions might not be the best starting point for making an academically appreciated research because there is a chance that a contribution to a scientific field might not be so relevant. To avoid that trap, we must analyse literature of the topic and gather some information gaps about cloud-based testing in the scientific field. Then we must make the constructive research the way that it provides information to fill the information gaps. (Iivari, 2013; Lehtiranta et al., 2016)

As Figure 1 demonstrates, the literature review and relevant background information for the study will be introduced first in the second chapter. After we have gathered enough background information, we will conduct interviews in the target team. The goal of the interviews is to understand the target team’s requirements for cloud-based testing. When
the requirements have been obtained, we start to collect empirical data and observations to proof how suitable cloud-based testing and partly simulated embedded systems are in the base station software testing area (Easterbrook, Singer, Storey & Damian, 2008). To observe data, we must develop few automated tests with the cloud’s test automation tool. Those tests are the constructs of this research. (Iivari, 2013; Lehtiranta et al., 2016)

Figure 1. Six principles of constructive research and how those are applied in this study. Figure modified from Lehtiranta et al. (2016).
2. Literature Review

This section will introduce the literature review which is carried out for obtaining knowledge about the topic. In the end, we will summarise conclusions from the literature to sum up the existing information and information gaps that should be filled.

2.1 Testing in General

When a large company continuously delivers software products for the customer, one release cycle process includes at least hundreds of software builds. When thousands of software developers develop same software simultaneously nightly, weekly or monthly, they need to modify it repeatedly and merge their changes with the latest build ready for release. In this study, the latest build is the software build which will be installed in the base station configuration. Then there are dozens of functionalities that should be tested including the software update. (Makki et al., 2016; Kumar & Mishra, 2016; Ghinea, 2016; Mathur & Malik, 2010; Mårtensson et al., 2016)

When changes in software builds are generally iterative, CRT brings a big improvement because it offers functionalities to execute the same test scenarios and test sets, for example, every day with the latest merged software build. Executing same tests every day requires that tests are automated because otherwise there is no time to run same tests manually when some tests require a lot of repetitions. The software maintenance is better when executing CRT to the software (Mathur & Malik, 2010). (Makki et al., 2016; Kumar & Mishra, 2016; Mårtensson et al., 2016; Karmore & Mahajan, 2016)

2.2 Testing Embedded Systems

We must remember that there can be different kind of SUTs and in this paper, we concentrate on large-scale software-intensive embedded systems (Mårtensson et al., 2016). Karmore & Mahajan (2016) state that testing has a lot of challenges in the embedded world mostly, because the real-time systems have to meet the challenge of assuring the correct implementation of an application not only dependent upon its logical accuracy but also its ability to meet the constraints of timing. To provide security and to implement real-time embedded system must be predictable because of the need to meet the timing requirements. (Karmore & Mahajan, 2016)

Karmore & Mahajan (2016) state that embedded systems require separate tools or separate code for testing like the case company has developed a lot of different testing tools. Presently, there is no mechanism available in the market which could reliably check a number of embedded systems under one platform. Testing is the most time consuming and costly portion of the embedded system development process (Karmore & Mahajan, 2016). These statements are hard to disagree when understanding that there can be dozens of different hardware configurations which must be tested. Testing different hardware configurations require many different hardware units and those can be expensive. Also, embedded system’s software code is complex and includes several different functionalities which must be checked.

When Karmore & Mahajan (2016) describe embedded system testing, it settles pretty much to the same area where the target team is focusing. Testing for embedded systems can be performed for different reasons like testing of the inputs or monitoring software
state and outputs for expected properties. For example, a test can verify that output such as an alarm is correct against the software specifications. Also, the output can save the system from a failure. For example, if software functionality breaks the software should perform some recovery action. This kind of verification and validation are important activities in the testing process. Verification is the process of human examination or review of the work product. Validations are required when there is need to execute the tasks on computers such as executing tests with testing tools. There are several types of testing which can be conducted on the embedded systems such as the white-box testing, black-box testing and exploratory testing. (Karmore & Mahajan, 2016)

2.3 Continuous Integration in Embedded Systems

Mårtensson et al. (2016) conducted a study which summarises factors that must be considered when implementing continuous integration in software development. Continuous integration is one of the Agile methods for developing software code continuously. The study also defines that continuous integration is widely promoted as an efficient way of conducting software development. The practice is said to contribute to that tests can start earlier. It also increases developer productivity.

What makes the paper interesting, is its objective which is embedded systems like this research’s SUT is. The study introduces problems that are faced when implementing continuous integration software development for large-scale embedded systems. We concentrate on things that are related to the testing part of the development process. (Mårtensson et al., 2016)

It seems to be a common problem that complex user scenarios need manual testing because those are very difficult to discuss in terms of automated testing. In addition to automated tests, the target team has manual tests such as tests where led colours in the hardware must be checked or other tests which require human attention. When continuous integration requires automated testing, it creates challenges if some tests must be done manually for many different hardware configurations because then there is no time to execute manual tests always for new software builds. (Mårtensson et al., 2016)

That leads to the situation that some continuous integration principles like “100 % of tests must pass for every build” cannot be applied completely. When all tests cannot pass for every new build, the rule should be replaced with other testing approaches. Also, as Mårtensson et al (2016) state, when there are dozens of different hardware configurations which require testing, it is not clear anymore what 100 % pass rate actually means – does it mean testing on all valid configurations or a representative subset. (Mårtensson et al., 2016)

When each test environment should have its own kind of hardware configuration, limited hardware becomes a problem. Using simulated environments is a common way to solve the issue related to hardware limitations. Simulated hardware does not require so much physical hardware but there are common problems mainly because testing with a simulated hardware cannot fully ensure that the same tests will pass for the integration build that runs on real hardware. (Mårtensson et al., 2016)

Also, Mårtensson et al. (2016) consider that any research that would mitigate the discussed impediments would be of great value in the embedded software development community.
2.4 Test Automation

When executing automated testing for embedded systems and other systems as well, it is always necessary to have a good test automation tool. At present, test automation is more and more mainstream which leads to the situation where many companies want to automate tests more and more – especially if software development follows continuous integration and other agile methods (Mårtensson et al., 2016). It needs to be noted that test automation might fail if it is not applied in the right context with the appropriate approach and the right test automation tool. Furthermore, many people in software business believe that it is not always wise to automate all tests. Properly implemented test automation can considerably decrease the cost of testing and increase the quality of the software but without plans and surveys test automation might fail. (Garousi & Mäntylä, 2016; Kumar & Mishra, 2016; Raulamo-Jurvanen et al., 2016)

A large literature review of Garousi & Mäntylä (2016) was analysed to gain more understanding about what issues matter when starting to implement test automation. When considering software related factors, one needs to consider maturity and stability of the software. This factor is relevant in this study because the SUT is a complex embedded system where new features and modifications appear frequently (Mårtensson et al., 2016). (Garousi & Mäntylä, 2016; Huang et al., 2012; Holappa, 2013; Roberts, 2016; Mårtensson et al., 2016)

Test related factors also matter when thinking about what and when to automate because planning tests might cost surprisingly much. However, when those tests have been automated and are possible to re-run, the cost decreases. In this case, it is important to think what kind of tests are wise to automate. As was mentioned in the beginning of this chapter, CRT should be automated when running same tests every day. However, some tests might need a lot of human judgement. For example, if analysing the results is hard and takes time, then it should be considered whether this is the best way to execute the test (Mårtensson et al., 2016). (Garousi & Mäntylä, 2016; Makki et al., 2016; Kumar & Mishra, 2016)

Test tool-related factors are important in this study when evaluating the new test automation tool. Using the wrong tool might be expensive and cause only a waste of money and time. For example, choosing a tool and implementing it for months might be a bad choice. One might realise afterwards that there could have been a better tool for those purposes. Tools should be studied and the impact on testing efficiency must be analysed from different perspectives to get the high-quality picture about the positive and negative sides of the tool. Criteria which matters when selecting the tool is issues like usability, functional suitability, maintainability, support, portability, and compatibility. Also, reliability, performance efficiency, security, support, and cost matter. (Garousi & Mäntylä, 2016; Raulamo-Jurvanen et al., 2016)

Human and organisational factors should be considered since test automation tool might need special skills like programming skills. Programming might be needed when implementing test automation so it should be considered whether the testers have enough programming skills. Before starting to implement tests, there should be enough training to use the tools. In this study, we should consider the quality of instructions to use a new test automation tool. Also, the new test automation requires investment and the benefits are not visible right after. It takes some time when benefits start appearing. (Garousi & Mäntylä, 2016)
Cross-cutting factors and other factors such as low automatability factor should be considered. Some tests might be difficult to automate properly or tools might not offer enough functions for it. A good test should cover also many scenarios as Garousi & Mäntylä (2016) mention. However, in this study, tests scenarios of tests are called test steps. All steps, which might bring up faults, should be tested. The software development process and the release model should be also considered when choosing test automation for the testing approach. (Garousi & Mäntylä, 2016)

2.5 Cloud-Based Testing

Even though test automation can be executed locally, it can be executed in the cloud as well. We should gather some general knowledge from a literature about cloud services and cloud-based testing to understand what we are analysing. Cloud computing was a new term about five years ago, but this year 2017, it is not that novel a term anymore. However, it is still in an upswing in the software industry. With a cloud computing, the user has ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. In this context, those resources are pre-initialised test environments. (Incki, Ari & Sözer, 2012; Raulamo-Jurvanen et al., 2016; Farooq & Quadri, 2016; Tao & Gao, 2016).

A new term for a cloud service delivery model has recently risen – Software Testing as a Service (STaaS) or Testing as a Service (TaaS), and Incki et al. (2012) use a term “on-line software testing” from cloud-based software testing. Cloud-based testing services seem to be a pretty much hybrid delivery model from basic delivery models Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) (Riungu-Kalliosaari, Taipale, Smolander & Richardson, 2016). Those delivery models are used when implementing cloud computing technology and architecture, to make software testing more flexible and elastic. Each company has different testing targets so cloud-based testing service should be implemented the way that a company demands. To understand cloud-based testing service, we should understand better those basic delivery models. Therefore, the next chapters will introduce IaaS, SaaS, and PaaS more closely. (Farooq & Quadri, 2016; Tao & Gao, 2016)

Cloud computing services have several different definitions for delivery models but SaaS, PaaS, and IaaS are widely adopted in the literature. SaaS means providing software application or services over cloud infrastructure for the consumer. The user can access those applications from various platforms through an easy-to-use client interface such as a web browser. PaaS, on the other, is behind an application. It enables consumers to deploy their solutions to the cloud by means of platforms such as application servers and database services provided by the Cloud Platform Provider. IaaS has been described as the lowest level of service model in cloud delivery models. In IaaS, the customer can acquire computing services and can deploy their own custom-configured systems in these resources potentially replicating their own existing infrastructures. (Incki, Ari & Sözer, 2012; Farooq & Quadri, 2016; Czernicki, 2011)

Tao & Gao (2016) introduce different approaches and infrastructures for cloud-based mobile testing. The study includes a lot of irrelevant information for this research, such as different cloud testing services to mobile applications. Even though that research is made for mobile testing, there is a lot of interesting information which is related to this study because that research deems cloud-based testing to solve the same kind of major issues. After formatting those issues suitable to this study, those issues include: (a) high costs on test infrastructures due to frequent changes and upgrades on SUT; (b) testing
complexity of complex embedded software; (c) large-scale test services, test simulations, and virtualizations.

2.6 Conclusions from Literature

Against the literature, we have one question which is following.

Sub-RQ1: What conclusions are there in the literature about cloud-based testing in general?

First, let’s start from executing tests to an embedded system that has dozens of different hardware configurations. Testing is difficult because embedded systems require so much hardware, which there is not enough to all test environments. Also, testing embedded systems can be time-consuming. Furthermore, there are problems in the embedded software development community which are related to implementing continuous integration in the embedded software development process. (Mårtensson et al, 2016; Karmore & Mahajan, 2016)

Cloud-based testing offers several major advantages which include: (a) offering large-scale testing anytime and anywhere; (b) cost reduction by sharing computing resources; (c) supporting on-demand testing services with elastic testing resources and (d) providing elastically scalable test automation. Also, Incki, Ari & Sözer (2012) state that testing in the cloud reduces costs because of installing and maintaining a test environment’s software doesn’t have to be a manual work. (Tao & Gao, 2016; Riungu-Kalliosaari et al., 2016; Incki, Ari & Sözer, 2012)

Even though cloud-based testing brings in many advantages, it might not fulfil its complete potential if the cloud’s test automation tool doesn’t work the best way. Furthermore, there are concerns about cloud’s security issues which should be investigated much more. (Tao & Gao, 2016; Riungu-Kalliosaari et al., 2016)

Cloud-based testing security issues should be studied more because security is one of the highly-valued requirements for effective cloud computing. Even though Riungu-Kalliosaari et al. (2016) studied cloud-based testing in their own research, they also encourage other researchers to conduct further empirical research about cloud-based testing. Interoperability testing needs more emphasis as a research area to ensure reliable service composition by means of integrating services from different service delivery models (Incki, Ari & Sözer, 2012). (Raulamo-Jurvanen, Kakkonen, & Mäntylä, 2016; Farooq & Quadri, 2016, Tao & Gao, 2016; Riungu-Kalliosaari et al., 2016)

Some researchers have studied cloud computing in different contexts such as small and medium-sized enterprises and the gaming industry. When considering that there is not much practical information about cloud-based testing in an embedded software development in large-scale companies, it would be useful to gather more information about that area. In the end, the base station software is the large-scale embedded software. (Riungu-Kalliosaari et al., 2016; Incki, Ari & Sözer, 2012; Farooq & Quadri, 2016; Tao & Gao, 2016)

Raulamo-Jurvanen et al. (2016) claim that cloud services as a test automation tool are not so popular in the software industry. That could be true but that doesn’t say how suitable the cloud could be as a part of a test automation. It is stated in other literature that cloud
services have increased popularity in the software industry and specifically in software testing (Farooq & Quadri, 2016).

Incki et al. (2012) observe that acceptance testing is an open research area for testing over the cloud beside test task management and interoperability testing because integrating cloud-based testing services from different service delivery models might contain compatibility problems. However, they published that statement in 2012 so we cannot say is it such an open research area anymore. Tao & Gao (2016) state that more research results are needed on cloud-based mobile testing. Results such as well-defined test models and coverage criteria are needed. Riungu-Kalliosaari et al. (2016) state that there is a need to observe real-life industrial experiences in order to develop a better understanding of the applicability of cloud-based testing.
3. Testing Workflow and Cloud Requirements

This section explains how current testing and cloud-based testing operates in the case company. The chapter’s last part presents the testing requirements which were gathered from interviews with the target team.

3.1 Current Testing Workflow

Figure 2 display how the test process currently operates. Garousi & Mäntylä (2016) introduce the same kind of tasks in their research where it was mentioned that before the tests are executed, they must be developed with a test environment’s computer. The main use cases include many sub-tasks so Figure 2 gives a too simple impression about the testing process. Use cases can be time-consuming because those require complicated tasks. For example, building a test environment can have many sub-tasks such as using the environment with different kind of testing tools and updating all software manually to tools (Pinola et al., 2013). Those things are closely related to same embedded system challenges as was noticed by Mårtensson et al. (2016) and explained in the previous chapter. As Figure 2 explains, Build Test Environment, Develop Test Scripts and Run Tests are a part of areas in which this study will concentrate.

![Main Use Cases in Current Testing](image)

**Figure 2.** Current use cases in a test automation.

**Build Test Environment**

Figure 3 demonstrates a test environment which must be build first so that tests can be developed and performed. A test environment contains a computer with needed testing tools which are needed for developing and performing tests. The computer should have
newest testing tools including the newest version of a test automation tool. The computer is connected to the SUT with an ethernet cable. SUT might have dependent components like test phones. A tester needs to do all initialization work for test environment by himself. Initialisation work contains, for example, updating the software for all testing tools. Following Figure 3 will show a test environment in a more specific context. (Garousi & Mäntylä, 2016; Karmore & Mahajan, 2016).

**Figure 3.** An example of a test environment which can also be called a test line.

The software is installed on the BBU hardware which will share an RRU specific software to RRU hardware. The software uses RRUs antennas for creating cells so that test phones can be connected to them. A big part of a test environment initialization is about finding a right type of hardware for building an SUT and updating the software (Pinola et al., 2013). Optical fibres are cables which are used to connect BBU and RRUs. Those cables offer the interface for BBU and RRU. There can be longer optical fibres and shorter ones. For example, a longer optical fibre might be 20 km long and the shortest is 2 m long.

**Develop Test Scripts**

After the test environment is ready, a tester can develop tests (Garousi & Mäntylä, 2016). As pointed out in the literature, a right test automation tool for developing tests should be carefully analysed because a test automation requires a lot of maintenance work. It is a waste of time to implement tests with an unusable test automation tool. That is the reason why test automation tool’s suitability should be analysed before starting to use it. (Kumar & Mishra, 2016; Raulamo-Jurvanen, Kakkonen, & Mäntylä, 2016).

Testers might need to create new test scripts or make modifications to existing ones. Also, debugging and testing scripts is necessary to see that tests work properly. For example, all unexpected alarms need to be detected during the test. The company has a lot of internal testing tools which show those alarms. Also, tools for gathering software log files should be used in every test. With tools, a tester can analyse why a test fails and send the logs to software developers. (Garousi & Mäntylä, 2016; Kumar & Misha, 2016)

Frequently updates come in to a test automation tool. Every modification to a script needs to be committed to SVN. With test automation tool, all scripts of the computer can be updated to the newest revision. It is tester’s responsibility to update tools always to the newest revision so tests will work as expected. An old version of the tool might not
understand some changes in the software code which cause that automated tests will not work properly with the newest software of the SUT.

Run Tests

When test scripts are developed, tests need to be performed for the SUT. If a test fails testers needs to analyse the reason for the failure before sending a fault report to software developers. The case company has internal tools for creating fault reports and collecting test results. (Garousi & Mäntylä, 2016)

3.2 Cloud-Based Testing Workflow

Now we know how the current testing basically works but the case company has provided an internal cloud service which follows the path of STaaS or TaaS (Farooq & Quadri, 2016; Tao & Gao, 2016). Inspired by studies of Tao & Gao (2016) and Mårtensson et al. (2016), we could describe it simulation-based embedded testing infrastructure on clouds which is required to have several key components, such as simulation controller and test connection manager.

![Figure 4](image_url)

**Figure 4.** This use case diagram shows how the testing process in the cloud works and what areas this study investigates.

Figure 4 displays main use cases that testers have when executing cloud-based testing. Use cases have changed a little when compared to current testing. Building a test environment is now reserving a test environment. Developing test scripts stays same but tests need to be implemented with different test automation tool. Running tests is still an important use case. The biggest difference between the previous and current testing is the
new use case which is creating a remote connection to the computer of the test environment. After the test environment has been assigned to a tester, the tester can create a remote connection to the test environment by using VNC. VNC offers a simple peer-to-peer architecture so that remote access for an unrivalled mix of Windows, Mac, UNIX and Linux computers is possible (Techspot, n.d.).

Figure 5. Cloud manager has hundreds of test environments.

As Figure 5 demonstrates, there are dozens of different test environments with different configurations. In this study, we will use only the test environments which have different baseband capacity because the target team needs configurations with different baseband capacity. One configuration might have more baseband capacity than other which means that it can simulate a bigger base station configuration. For example, Test Environment A has the smallest baseband capacity and it cannot have more than three cells. If a tester needs a base station configuration with nine cells, Test Environment B should be reserved because it has more baseband capacity. When there is more baseband capacity, the base station can have a bigger hardware configuration with more active cells where a cell phone can attach (Wannstrom, 2013). Also, a tester must select a software build when creating a reservation. Then he or she just waits for his or her own turn.

Figure 6. An example of a test environment in the cloud.

As Figure 6 shows, a single cloud test environment contains partly-simulated SUT. BBU is not simulated but RRU hardware is. As was introduced in current testing, there might be several RRUs for one Base Station configuration. Now, instead of the real RRU, there is an RRU simulator which can simulate many RRUs. The RRU simulator might simulate many RRUs depending on the XML file it has. Information in XML file defines the
hardware which is simulated. A tester can change XML file and fill in there those RRUs which must be tested.

Figure 7. This process diagram demonstrates required activities when developing tests in the cloud.

Figure 7 contains the process diagram which summarises the whole test environment reservation process. A tester uses the web application to create a reservation for a needed test environment. The reservation must include the information such as the needed software build and test automation tool’s version. When the environment is ready to use, the tester gets an email or SMS notification which contains information of the test environment. With the remote connection, a tester gets a computer desktop for carrying out activities such as developing and debugging test scripts.
The process diagram of Figure 8 display how already conducted tests are driven in the cloud. Tests can be run through the web application where the system automatically reserves a right type of a test environment following the information that a tester provides. Also, it is possible that tests are driven automatically every week or every day. If a tester wants to schedule test runs that way, it is possible by adding right tags to Robot Framework’s test suite. Then the system automatically searches tests suites with those tags from SVN and runs tests according to the information of tags. By using the tags, a tester doesn't need to do anything else than analyse the failed tests by using the reporting tool.

3.3 Cloud Requirements

This chapter analyses the target team’s requirements for test automation in the cloud. Those requirements will be summarised from qualitative data which will be collected by interviewing testers from the target team.

3.3.1 Interview

Let’s consider the first research questions to which this chapter gives answers.

*Sub-RQ2: What requirements do embedded system testers have for cloud-based testing?*

There were two options for collecting a data for the question. The options were an interview or a questionnaire. In the end, the choice was the interview. Executing the
interview makes possible to explain more about the topic to interviewees who don’t have so much understanding about the target tool. Collecting data about concerns and possibilities needed a small introduction about the cloud service before asking questions. If an interviewee did not have any thoughts to a question such as possible disadvantages in the partly-simulated environment, the interview made it possible to explain the question to the interviewee more thoroughly and thereby obtaining more data.

In interviews, all kind of concerns and possibilities regarding all areas of the cloud service were collected. Those areas include test environments, the cloud’s web application, and the test automation tool and possibly other issues that would come up. Furthermore, the data regarding possibilities that testers see in the cloud service was collected. It could have been possible to ask directly for the requirements instead of asking questions about concerns and possibilities. However, this way a good deal of information was collected to the data sheet. The interview was an un-structural since with that type of an interview, it is more effective to collect information and testers own thoughts were heard.

A plan was to conduct the interview for all 15 testers from the target team. Interviewees were chosen from the target team because most of them have many years’ experience of the base station software testing. According to the study of Baker, Edwards, & Doidge (2012), there is not a right number of interviews for a valid research. It depends on how big part the interview is for the research, how much time there is for interviewing and how versatile answers there need to be in order to make conclusions (Baker, Edwards, & Doidge, 2012). When the interview was done to 9 testers, including a couple of testers having special tasks like developing test automation and test planning, it became clear that the same thoughts came up and there was already plenty of data for the research. Last interviews were cancelled because of the tight schedule of the research.

3.3.2 Interview Analysis

All 9 interviews provided a lot of answers. Main answers were written to notes during the interview. Each interview contained an own note paper. Afterwards, the notes were listed on the data sheet. There were different kinds of answers which related to the same issue so they were combined into a single requirement. Several answers to each requirement were calculated. For example, all 9 testers mentioned that partly-simulated SUT must support a different kind of configuration features such as big configurations containing many RRUs. Because the cloud-based service includes four different areas, the requirements are divided into four categories where next chapters will focus. Following Figure 9 visualises those requirement categories.
Requirements 1 contain all test environment related factors including a test environment reservation and remote connection to the test environment. Requirements 2 contain all the test automation tool related factors which are related to test automation development. Requirements 3 contain the web application related requirements. There were not so many requirements in that category because it only contains functionalities which are related to creating a single test run. There are also Requirements 4 which are related to fault detection capability of partly-simulated SUT. Partly-simulated SUT is a part of a test environment so those could be also considered as a part of Requirements 1. However, the Chapter 7. investigates fault detection with the partly-simulated SUT so we place those to a category of their own.

Requirements 1: Test environments

Chapter 4. will analyse main requirements for requirements which are related to test environment reservations of the cloud. There was one requirement which was most often mentioned in interviews. The requirement was “Support for different configuration features” which is related to the fact that there are dozens of different SUT configurations which include several hardware components that should be tested. There is a big question how well partly-simulated SUT can actually simulate all kinds of hardware. Components such as different optical fibre lengths belong to base station configurations. Testers said that the cloud should at least offer a possibility to test big configurations such as 18 cells configuration.

“SUT should be revived effectively after it crashes” was not seen as a big problem. It was mentioned only once. Maybe the reason why “SUT should be revived effectively after it crashes” was not mentioned that often was the fact that testers don't need to care about it so much. There are automatic scripts which initialise the test environment when something crashes or a tester releases the reservation. Initialisation contains all software installation from a computer’s software to the SUT’s software.

Four requirements were mentioned four or five times and were settled to the middle category. First of them was “Test environment reservations should be reliable” which
means that reservations should not break suddenly and there shouldn’t be remote connection problems. The second requirement was “Reservation waiting times shouldn’t be too high” which simply means that testers shouldn’t need to wait for a test environment too long. That might be the biggest issue when considering the literature which was studied in the previous section, which explained that the cloud testing service should make testing more flexible and elastic. The cloud service is not flexible enough if building manually a test environment is faster than reserving it from the cloud.

“Test environments should be initialized automatically” is closely related to the previously introduced requirement “SUT should be revived effectively after it crashes”. Always, when the tester gets a test environment, it is initialized automatically with same automatic scripts that are used for reviving a crashed environment. Also, a requirement “Should be able to simulate different RRU types” is closely related to the requirement “Partly-simulated SUTs should support more configurations” because different RRU types are a part of different base station configurations.

Table 1. Requirements 1: Test environments

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Mention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test environment reservations should be reliable</td>
<td>4</td>
</tr>
<tr>
<td>Reservation waiting times shouldn’t be too high</td>
<td>5</td>
</tr>
<tr>
<td>Partly-simulated SUTs should support more configurations</td>
<td>9</td>
</tr>
<tr>
<td>SUT should be revived effectively after it crashes</td>
<td>1</td>
</tr>
<tr>
<td>Test environments should be initialized automatically</td>
<td>5</td>
</tr>
<tr>
<td>Should be able to simulate different RRU types</td>
<td>4</td>
</tr>
</tbody>
</table>

Requirements 2: Test automation tool

When a tester starts to automate tests, test scripts need to be written. The cloud has a different test automation tool than the tool which the target team currently uses. In the interviews, the new tool caused a lot of conversation. “Tool should have good documentation and instructions” was mentioned five times, which is understandable because without good instructions and test examples it takes the time to remake all automated tests. Then most of the time goes to learning.

Requirements “Test scripts maintainability should be good” and “Other teams’ test resources should be accessible” was mentioned four times. Maintainability should be good because of the software changes frequently and then it is required to modify test scripts. In the case company, the cloud’s test automation tool has been promoted much. It’s been said that other teams’ test libraries and resources are easy to use in own tests. However, there were many doubts about how well that works.

Requirements “Should be possible develop comprehensive tests” and “Should be possible to automate current manual tests” were mentioned three times. Comprehensive tests mean that the test should have real-time checks. For example, all unexpected alarms should be founded even if the test step is checking different things like cells states. The target team would like to automate at least data call tests. That manual test would be needed to automate. However, there are doubts that all faults won’t be found because of the cloud’s partly-simulated SUT.
Two requirements were mentioned only twice and interviewees did not see those as big problems. Requirements were “Changing hardware configuration for simulator should be easy” and “Automation tool should work with other testing tools”. Basically, configuration change happens by modifying simulator’s XML file. From example, tester writes all needed RRU type information to that file. Also, a tester writes to the file so many RRUs than SUT should simulate. When developing tests with the new tool, current testing tools should also be accessible. For example, software logs should be collected during automated tests. There is an internal tool for that purpose.

Table 2. Requirements 2: Test automation development

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Mention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test scripts maintainability should be good</td>
<td>4</td>
</tr>
<tr>
<td>Other teams test resources should be accessible</td>
<td>4</td>
</tr>
<tr>
<td>Changing hardware configuration for simulator should be easy</td>
<td>2</td>
</tr>
<tr>
<td>Should be possible to develop comprehensive tests</td>
<td>3</td>
</tr>
<tr>
<td>Should be possible to automate current manual tests</td>
<td>3</td>
</tr>
<tr>
<td>Tool should have good documentation and instructions</td>
<td>5</td>
</tr>
<tr>
<td>Automation tool should work with other testing tools</td>
<td>2</td>
</tr>
</tbody>
</table>

Requirements 3: Test executions

The web application of the cloud did not draw so many requirements than other two areas. Four testers mentioned that the web application should be easy to use. It means that tests should be able to be executed easily with the application. Also, test results should be analysed easily. Four interviewees responded that the application’s learnability should be good. It should have good instructions.

Two testers saw the tool’s reliability as a concern, which is understandable. If the website doesn’t work, then tests cannot be performed with the cloud’s test environments. There were two requirements which were mentioned only once. One was that tests should be able to be executed in any place such as at home. The other was that the web application should work with the reporting tool. When tests are executed with the web application, test log files should be easy to analyse and fault report should be easy to send forward to software developers.

Table 3. Requirements 3: Text execution through the web application

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Mention</th>
</tr>
</thead>
<tbody>
<tr>
<td>The web application should be reliable</td>
<td>2</td>
</tr>
<tr>
<td>Learning to use the web application should be easy</td>
<td>4</td>
</tr>
<tr>
<td>Testing or retesting should be easy</td>
<td>4</td>
</tr>
<tr>
<td>The web application should be accessible everywhere</td>
<td>1</td>
</tr>
<tr>
<td>The web application should work with the reporting tool</td>
<td>1</td>
</tr>
</tbody>
</table>
Requirements 4: Partly-simulated SUT

First, these following requirements were a part of Requirements 1 but it was decided to place those in this own category because Chapter 7 focuses only on these requirements and Requirements 1 are analysed in Chapter 4.

“Fault detection with partly-simulated SUT” was the requirement which was mentioned several times. It is related to the fact that when using simulated RRUs all bugs might not be found which was also pointed out as a common way in the literature by Mårtensson et al. (2016). If bugs are not found before the software is released, it is a customer who finds them which is never a good thing.

“Simulator's software shouldn't cause new faults” was not seen as a big problem in the interviews. However, new faults might be problematic even though it was only mentioned once. If the simulator’s software has own bugs, then there is another software which needs to be tested. It can cause more work which is not a good thing.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Mention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault should be detected with partly-simulated SUT</td>
<td>8</td>
</tr>
<tr>
<td>Simulator's software shouldn't cause new faults</td>
<td>1</td>
</tr>
</tbody>
</table>

3.3.3 Summary

Interviews provided a lot of requirements which can be used when analysing the cloud service. What is interesting to see is that the web application of the cloud service did gather only a few of requirements. Most requirements are for test automation tool and test environments in the cloud.

These findings support the research of Mårtensson et al. (2016) which also stated that simulated SUT might not find all same faults that the real SUT. Also, large-scale software-intensive systems like these 4G base stations include so many different hardware configurations that there is a real problem because there need to be so many different test environments to perform continuous integration testing reliably. (Mårtensson et al. 2016)

Furthermore, according to literature, test automation tool needs a comprehensive analysis process before it is possible to actually start to using it in the field because using a test automation tool which is not usable might be a waste of time and other resources (Garousi & Mäntylä, 2016; Kumar & Mishra, 2016; Raulamo-Jurvanen et al., 2016). These interviews support those findings because there were so many concerns related to the cloud’s test automation tool.
4. First Step: The Test Environment Reservation

Using the cloud’s test environments gave a lot of positive experiences. This chapter will introduce those results more closely.

4.1 Use Case Overview

Let us remember use cases where this study concentrates on. As Figure 10 shows, we start to investigate how test environment reservation and remote connection work. Those are first use cases which need to be conducted before test automation development can start. We will collect empirical data from several different reservations for analysing Requirements 1.

![Use Cases of Cloud](image)

Figure 10. Use cases of test environment reservation and remote connection creation.

During interviews, it became clear that testers have concerns about waiting times for receiving test environments from the cloud. Furthermore, testers did have concerns about the reliability of reservations: some interviewees believed that reservations can break suddenly or connection problems to test environments can occur. Sub-RQ3 states briefly the issues to which this chapter answers.

Sub-RQ3: Do the test environments of the cloud fill the requirements?
4.2 Waiting Times in the Cloud

First, let us sum up the reservation process. A tester can make a test environment reservation through the cloud’s web application. There one needs to fill in the information that defines what kind of an environment is needed. The information contains a test environment type, SUT’s software build, a simulator’s software build, a test automation tool’s revision and the reservation’s duration. A maximum duration can be four hours but a tester can extend the reservation during the last hour. The maximum extension can be three hours.

There are different types of test environments in the cloud. In this research, we collected a data of reservations which were made for three different type of test environments. We call those types Test Line A, Test Line B and Test Line C. During the research process, we collected waiting times for 55 different Test Line A reservations, 45 different Test Line B reservations and 19 Test Line C reservations. The reason why Test Line C has so few reservations is the fact that the cloud contains only five of them. For example, the cloud has about 60 Test Lines B.

![Figure 11. Waiting times for Test Line A.](image-url)
Figure 12. Waiting times for Test Line B.

Figure 13. Waiting times for Test Line C.
As Figure 11, Figure 12, Figure 13 and Figure 14 displays, a tester gets the environment faster if the reservation is made early in the morning. As the scatter diagrams show, almost all reservation waiting times are under 50 minutes until half past nine. After that waiting times will increase drastically. Then the tester must prepare to wait for the environment over 150 minutes which is a long time to wait for having a test environment.

Maximum waiting time is about 200 minutes with Test Line A. It is a long waiting time for a tester. When we remember that testers had concerns towards waiting times this data is important for making the conclusion that test environments are not quickly available. If a tester reserves a test environment before half past nine, then waiting times stay most probably under two hours which is quite a long time as well. However, if the reservation is made after half past nine, waiting time will be most probably from one hour to three hours which is a really long time.

4.3 Reservations Workability

Considering the reservations workability which is related to requirements of reliability and initialization efficiency, we must analyse reservations against requirements. If there was connection problems or reservations ended before the duration time ended, then the reliability requirement will not work properly. Also, if it was necessary to update any software to test environment manually, then initialization requirement will not work properly.

Connection failures mean how frequently a remote connection broke down or has any interruptions. When developing tests, 38 reservations were made in the cloud. During those reservations, connection failures were visible few times when the remote connection broke a little or was lost completely. However, the remote connection was easy to recreate. Connection problems were annoying and took some patience. From these observations, we can say that there are connection problems.

Next, we must consider the reliability more from the point of view of how many times a cloud system cancels the reservation before the maximum duration was ended. Those reservation endings were never seen because the researcher was always able to use the
environment as long as was needed. When reservation’s ending was near, there came a lot of notifications that the reservation will end soon. Notifications appeared in email and as a popup window to the remote computer. If maximum four hours wasn’t enough, the reservation was easy to extend as many times as was needed. Maximum hours for extension was three hours.

Finally, initialization efficiency was possible to analyse. In those 38 reservations, all software was installed automatically which was maybe the most satisfying thing when using the cloud’s test environments. There was no need to update manually any software to test environments because all necessary software was already installed. However, if testers need to use some old software it might not be available automatically to the cloud’s test environments. Then a tester needs to update a software manually to the remote computer.

4.4 Data Against Requirements 1

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test environment reservations should be reliable</td>
<td>Fail</td>
</tr>
<tr>
<td>Reservation waiting times shouldn’t be too high</td>
<td>Fail</td>
</tr>
<tr>
<td>Partly-simulated SUTs should support more configurations</td>
<td>N/A</td>
</tr>
<tr>
<td>SUT should be revived effectively after it crashes</td>
<td>N/A</td>
</tr>
<tr>
<td>Test environments should be initialized automatically</td>
<td>Pass</td>
</tr>
<tr>
<td>Should be able to simulate different RRU types</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Considering the gathered empirical data, we can reliably answer half of the Requirements 1 as Table 5 shows. Because of detected remote connection problems, we can say that the requirement “Test environment reservations should be reliable” will not work properly. Also, we can say against the requirement “Reservation waiting times shouldn't be too high” that it will not work properly either because there were so many more reservations which had to wait many hours than reservations which had to wait an hour or less. However, the requirement “Test environments should be initialized automatically” was accepted because test environments were always initialised with comprehensive tools and software.

There are three requirements where we have not enough empirical data so we cannot reliably say that those would be accepted or not accepted. However, the researcher has concerns that the requirement “Partly-simulated SUTs should support more configurations” will not be accepted because it was noticed that hardware configuration with more than 9 cells was not possible to simulate.

The requirement “SUT should be revived effectively after it crashes” was not analysed because this research focused only on using the cloud’s test environments and not maintaining them. Basically, the requirement means the situation when automatic initialization process won’t work. For example, the hardware should be fixed manually if it has crashed. That might need some future work to see how much work it takes to maintain test environments in the cloud.
In the end, there was the requirement “Should be able to simulate different RRU types” where we have not enough empirical data. However, during tests implementation process, that kind of RRU types which any hardware configurations are currently using were simulated, which encourages to say that different RRU types can be used.

These cloud problems, excluding the test environment initialization, were not discussed in the literature. In fact, in the literature it was only explained a little how cloud-based testing actually works. For example, in the literature it was not discussed how the use cases of cloud-based testing concretely go.
5. Second Step: The Test Development

This section will introduce findings during tests implementation process. We will introduce test automation tool before we start to describe different findings.

5.1 Use Case Overview

The previous chapter introduced the results of a test environment reservation. As Figure 15 demonstrates, a tester can start to develop tests through the remote connection after the test environment has been received.

![Use Cases of Cloud Diagram](image)

**Figure 15.** Develop tests is the second use case after the remote connection is established.

From interviews, we gathered *Requirements 2* which consider test development with test automation tool. Requirements will be analysed during tests implementation process as the *Sub-RQ4* defines. Tests are “Base station reset”, “Cells block and unblock”, “Base station block and unblock” and “Simple data call”. Later in this chapter, we will explain the content of tests more specifically.

*Sub-RQ4: Does the test automation tool of the cloud meet the requirements?*

5.2 Test Automation Tool

First, we will look a little closer at the test automation tool which was used for implementing tests. There has been a survey which represents that Robot Framework is one of the most popular open-source test automation tools together with Selenium (Raulamo-Jurvanen, Kakkonen, & Mäntylä, 2016). Also, the cloud’s test automation tool has been built around Robot Framework. Robot Framework is a generic test automation
framework for acceptance testing and acceptance test-driven development (ATDD) and it utilises the keyword-driven testing approach. (Robot Framework, n.d.)

The company’s tool includes test libraries and resource files which both contain keywords that can be used in a test file, more commonly known as a test suite. A test suite file will be run by a terminal command when tests need to be executed. An example test suite file’s code can be seen below. The code is just a simple example which demonstrates that “First test” has three keywords, which can be considered as test steps. Those keywords are “Create Directory”, “Create File” and “File Should Exist” and usually they can take some arguments like in this case some directory paths. The functionalities to those keywords are defined in either the library named OperatingSystem or the resource file named SomeResource.robot. (Robot Framework, n.d.)

```
*** Settings ***
Library       OperatingSystem
Resource      SomeResource.robot

*** Test Cases ***
First test
  Create Directory  ${TEMPDIR}/dir
  Create File       ${TEMPDIR}/dir/temp.txt  The File Contents
  File Should Exist ${TEMPDIR}/dir/temp.txt
```

Libraries and resources can be thought of as a header file in an object-oriented programming language. The tool offers ready-made libraries and resources that can be used when creating test suites. Even though the tool contains ready-made libraries that are free to use, it is possible to create one’s own resource files that act like libraries. A user can generate common keywords to those files so there is no need to use same keywords in different test suite files. (Robot Framework, n.d.)

![Figure 16](image.png)

**Figure 16.** Robot Framework’s higher level architecture contains different file libraries and components. The figure is borrowed from a user guide of Robot Framework (n.d.).

Robot Framework architecture contains many factors in different levels as Figure 16 shows. Those are test data, Robot Framework, test libraries, test tools and an SUT. Test
data structure should be readable and easy to edit for different testing purposes. The user should generate test data before Robot Framework is started because it processes test data, executes test cases and generates logs and reports. The core framework doesn’t know anything about the SUT that are used for testing the system. The interaction between a test data and the SUT is executed by test libraries which can either use application interfaces directly or use lower level test tools as drivers. (Robot Framework documentation, n.d.)

5.3 Test Cases Implementation

This chapter explains implemented test cases and how the findings from implementation process compare against the Requirements 2.

5.3.1 Test Cases

The target team has many test cases which should be tested with different SUTs but this research concentrates only on four of them which are shown in Table 6. The reason why these tests were chosen was that tests include simple functionalities that should work in many tests. Also, the reason why simple data call test was chosen, was the fact that currently it has not been automated completely because of the lack of test phones. Mathur & Malik (2010) define a test case in the following way.

“A test case is a set of conditions or variables under which a tester will determine whether an application or software system meets its specifications at the unit level”

These test cases include a lot of testing principles which are related to embedded software testing. It must be verified that multiple systems outputs work correctly when some inputs are performed. (Karmore & Mahajan, 2016)

Table 6. Test cases where this study concentrates on.

<table>
<thead>
<tr>
<th>Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base station reset</td>
</tr>
<tr>
<td>Cells block and unblock</td>
</tr>
<tr>
<td>Base station block and unblock</td>
</tr>
<tr>
<td>Simple data call</td>
</tr>
</tbody>
</table>

Test Case 1: Base station reset

As the name indicates, this test restarts the base station just like a computer can be restarted. It is a common test where faults might show up – especially if the SUT is a large configuration which has many RRUs or cells. For example, cells might not reach on air state after the base station starts up because the system can fail to configure cells. Also, some timing problems between software components might emerge (Karmore & Mahajan, 2016). After the base station starts up and all cells reach on air state, calls should be verified with a test phone. The test phone should be connected to the required cell and throughput of data transfer should be high enough such as 10 Mbps.
From this description, we can collect many inputs and outputs. Main input of this test is triggering the base station reset. Then there are several outputs which must be verified, for example, there should not be any unexpected alarms after the reset. Also, all cell states should be in operational state. Those are the main things of the test even though there can be many other unexpected situations, for example, RRUs might be missing after the reset. (Karmore & Mahajan, 2016).

**Test Case 2: Cells block and unblock**

In short, this test’s input is to block all cells of the base station. Then the correct output is that cell states are blocked. If states are not blocked, a software bug is probable. Cells shouldn’t be in the operational state anymore and data calls cannot be performed. After cells are blocked correctly, they must be unblocked. It must be checked that cells are in an operational state without any unexpected alarms. After cells reach on air state, data calls should work with all cells.

**Test Case 3: Base station block and unblock**

This test is quite like the previous test case but it blocks the whole base station including base station hardware and cells. When the base station blocking has been executed, any cell shouldn’t be in the operational state anymore and base station blocked alarm should be visible without any other alarms. After unblocking the base station, the software executes softer reset than previously introduced base station reset. The base station should reach on air state in three minutes. When cells are on air, data calls should be tested as in previous tests.

**Test Case 4: Simple data call**

This test should be performed after each test but it is an own field because currently calls are not completely performed in test automation due to lack of test phones. Here we have a concrete example that Mårtensson et al. (2016) also pointed out. Embedded system testing requires too much of hardware or equipment so simulators might be a good solution. Partly-simulated SUTs offer the possibility to automate data calls so if this test can be automated, it is a big advantage against the current situation.

5.3.2 Findings Against Requirements 2

This chapter introduces findings when implementing tests to the cloud. Findings are analysed against Requirements 2. Each related requirement is gone through one by one by explaining how the process worked against the requirement.

**Requirement: Test scripts maintainability should be good**

A lot of positive observations were detected considering tests maintainability because every keyword was made only once. After the keyword was made to own resource file, each different test suite file could use same keywords. That is useful because it makes possible to recycle same resources for every test suite. There can be many different test suites which all have the same test step like the base station reset. Tests can be modified easily in situations where tests will not work due to the changes in software functionalities. For example, a situation where none of the tests worked with a new software build came up. In that case, it was easy to make a modification to the keyword where the problem was. Next time all those test suites which used the modified keyword worked.
**Requirement: Other teams test resources should be accessible**

In the company, it was implied that test automation tool offers a big advantage because all testing teams can use each team’s resource files. It should save time when everybody does not have to recreate test steps. However, soon it became clear that it did not work so well because usually, other team’s keywords did not work directly in own tests. Then it was necessary to copy keywords to own resource file where the modification was done. Main reason was that other team’s tests were not so comprehensive as the target team demands. For example, the keyword which checked different cell states was not comprehensive enough because the keyword had only one state check even though the cell might have almost ten different states which should be checked.

Also, the tool has libraries which contain ready-made keywords. Those libraries have a better maintenance. However, keywords in libraries are not complete test steps so those can be used only in resource files where a test step was created completely. In that case, test development still requires a lot of work without recycling.

Clearly, the tool’s resource files need a better maintenance so that using them is possible without questioning whether they work as expected. There is always a chance that the target team can develop own resources and maintain those but then we cannot talk about recycling tests of others. However, it should be remembered that when tests are developed, maintaining does not require so much time as the literature also pointed out (Garousi & Mäntylä, 2016).

**Requirement: Should be possible to develop comprehensive tests**

Developing comprehensive tests was possible with certain restrictions. Tests “Base station reset” and “Base station block and unblock” included restrictions because it was not possible to verify start-up time reliably. The simulator makes start-up times much bigger than the start-up time actually is because simulator does not raise cells up automatically in cooperation with the SUT. The simulator needs to be started with an extra command before cells are in the correct state. That command is not needed when testing the real SUT because cells automatically reach on air state after the start-up. Also, it took time to get “Base station reset” and “Base station block and unblock” working because the simulator was stopped during each reset. Then we found out that simulator needs the extra command after the reset. This is also related to the statement of Mårtensson et al. (2016) about reliability issues of simulated hardware configurations.

However, “Cell block and unblock” and “Data call” were possible to automate almost comprehensively because those tests did not require the base station reset so the simulator remained in progress. The only step which was not done was the transmission power. “Data call” test was automated even though it was not currently automated. This can be considered to increase test coverage.

**Requirement: Changing hardware configuration for simulator should be easy**

Also, executing same tests with different hardware configurations increases test coverage. Simulated hardware configurations will be defined by changing XML files and performing same tests in many different hardware configurations requires that different configuration files are created for different test suites. In the beginning of test suites, there is a test setup keyword which put pre-created configuration files to the system. When test setup is performed, the system simulates the hardware configuration that configuration files orders. Configuration files are a part of SVN repository where test suites also are.
As many test suites for the same test as there are configuration files have to be created. The simulator did not support to simulate the configuration which has more than nine cells. That is a limitation when considering the target team’s demands because they must test bigger configurations.

**Requirement: Should be possible to automate current manual tests**

As was already mentioned, “Data call” test was automated even though it was not currently automated because there are not enough test phones to automate it. Real test calls would require so many phones that they might never be available. It was a great finding which will give a big advantage to the tool. Even though testing simulated hardware configuration is not always reliable, simulation offers possibilities to test functionalities that cannot be tested with real hardware because that would require too much hardware or equipment (Mårtensson et al. 2016).

**Requirement: Tool should have good documentation and instructions**

From the researcher's point of view, the instructions for the tool were too optimistic because the documentation gave an impression that the tool should have ready-made keywords for tests. The task really progressed in the way that other team’s keyword was copied to own resource file where the modification was done. That seemed redundant task for a researcher even though there were comprehensive instructions for developing tests for simulated configurations.

It must be pointed out that learning to use simulated test environments and developing tests took a lot of time and required a lot of understanding for Linux systems of test environments. For example, using Linux terminal window was necessary when debugging tests. When considering these findings, we should remember the statement from Garousi & Mäntylä (2016), that testers’ skills should be considered when starting to use a tool. That notification has much more meaning now when understanding how complex test development is with the target tool. If testers don’t have enough skills to use the new tool, it might not be that good of an option.

**Requirement: Automation tool should work with other testing tools**

Basically, this requirement demands that automated tests can use same testing tools that are currently used. For example, executing base station reset requires that the reset command is set from the base station controlling tool. The test suite must have a keyword that makes the connection to the tool and triggers the reset. Also, there are other tools for observing different states and alarms or taking software log files from the test. Test suites must use those tools so that test steps are possible. When these four tests were developed, there were no problems with this requirement. All necessary tools were accessible so this requirement worked in these situations.

**Summary table**

Table 7 sum up the previous findings against Requirements 2.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test scripts maintainability should be good</td>
<td>Pass</td>
</tr>
<tr>
<td>Other teams test resources should be accessible</td>
<td>Fail</td>
</tr>
<tr>
<td>Changing hardware configuration for simulator should be easy</td>
<td>Pass</td>
</tr>
<tr>
<td>Should be possible to develop comprehensive tests</td>
<td>Fail</td>
</tr>
<tr>
<td>Should be possible to automate current manual tests</td>
<td>Pass</td>
</tr>
<tr>
<td>Tool should have good documentation and instructions</td>
<td>Fail</td>
</tr>
<tr>
<td>Automation tool should work with other testing tools</td>
<td>Pass</td>
</tr>
</tbody>
</table>

### 5.3.3 Real Time Fault Catching

The target team’s test automation developer suddenly told one important requirement for the test automation which was a real-time fault catching. It means that when the test is ongoing, all unexpected situations should be noticed in every test step. For example, if the test suite is going on the line where the base station reset is triggered, some RRU missing fault might not be detected. One of the biggest problems when creating tests with unified test automation tool was related to this requirement.

When creating first three test cases it took a long time to find a useful keyword which made possible to check all log messages from the test. That was the best way to solve the problem. The keyword catches faults from logs in the end of the test. The keyword doesn’t catch unexpected alarms in a real time but at least it catches unexpected alarms which are the most important thing in this case.
6. Third Step: Test Runs in Cloud

In this chapter, we concentrate on running tests in the cloud. After the introduction, we summarise the findings against the research question.

6.1 Use Case Overview

As Figure 17 presents, test suites can be driven through the cloud’s web application after we have implemented those.

![Use Cases of Cloud Diagram]

**Figure 17.** Run test is next use case after tests are developed.

The following *Sub-RQ5* was generated from the interviews because those provided some requirements that the web application should fulfil when tests’ execution are performed.

*Sub-RQ5: Do the test runs in the cloud fill the requirements?*

We analyse the requirements from which we have empirical data and observations. The first requirement was that the “The web application should be reliable” which means that tests are performed when a tester triggers them. The second requirement was that “Learning to use the web application should be easy”. Testers should understand easily how to use the web application’s graphical user interface. The third requirement was that “Testing or retesting should be easy” which means that executing tests is fast and analysing test is easy. The fourth requirement was that “The web application should be accessible everywhere”. The last requirement was that “The web application should
work with the reporting tool” meaning that test results should be analysed with the reporting tool that the company uses.

6.2 Findings Against Requirements 3

This chapter provides answers against Requirements 3. Table 8 sum up answers which are explained in this chapter.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>The web application should be reliable</td>
<td>Pass</td>
</tr>
<tr>
<td>Learning to use the web application should be easy</td>
<td>Pass</td>
</tr>
<tr>
<td>Testing or retesting should be easy</td>
<td>Fail</td>
</tr>
<tr>
<td>The web application should be accessible everywhere</td>
<td>N/A</td>
</tr>
<tr>
<td>The web application should work with the reporting tool</td>
<td>Pass</td>
</tr>
</tbody>
</table>

6.2.1 Test Runs in the Cloud

When creating a test run, testers need to fill in same information that was required when reserving a test environment because the cloud system needs the test environment where it can execute the test. However, it also requires a path to the test suite file in SVN repository. After the test has been created, the cloud service automatically reserves a test environment, installs the correct software to it, runs test suite in it and reports test results to the reporting tool. It was easy to learn from the researcher’s point of view because it did not basically require anything else than filling in the information. That is the reason why requirement “Learning to use the web application should be easy” is accepted.

It was satisfying from the researcher’s point of view to see how easily tests can be executed with the web application. A tester only creates a test run, then waits for results and analyses the results. Basically, the tester does not need to bother about the test environment at all. The environment is in the cloud and it is always initialized with the original settings and a right software. Then the system runs the test in the environment. A tester gets a link to test results on the website. Results contain all downloadable log files which were generated during the test. However, executing tests was not so fast which will be analysed closely later. Because of those slow execution times, the requirement “Testing or retesting should be easy” did not work. Sometimes retesting should be performed immediately and cloud does not make that possible.

Also, there was the requirement that “The web application should be accessible everywhere “. For that requirement, we don’t have any supportive data because all tests were executed in the same office. Thus, we did not actually test it. However, because the platform is the web application it should be possible from anywhere if the network supports the application. However, the last requirement “The web application should work with the reporting tool” was true because all 34 tests were analysed with the reporting tool successfully so we can say that the cloud’s web application works with the reporting tool.
Indirectly one test architect made a request that the cloud should support stability and load tests which might last over 24 hours. After a little investigation from the cloud’s documentation, it was found that test runs have timeouts in the cloud. Tests which should be performed every week can last only 12 hours and tests which should be performed every day can last only 3 hours so the cloud is not suitable for long stability and load tests. However, those tests are so rare that it is not such a big disadvantage. Probably those rare load and stability tests can be executed for example with the current automation tool.

6.2.2 Waiting Time Before the Test Starts

It was already discussed in the previous chapter that the cloud does not offer fast test execution. In this chapter, we have some data to proof that statement. In Figure 18, we analysed test execution times of 26 cloud tests. All those tests were executed for test suites which were developed previously. In the figure, we can see the data of test runs containing a test run creation time and the waiting time before the tests were started.

We would have also liked to know how much time the whole test process takes from a creation time to finishing a test run but there it depends on how a test has been made and what test suite is used. If a test suite is a complicated and takes time, then execution time is longer. So, it is better to analyse how much time the cloud tool needs to get a test environment ready before the test can be started. That clearly is the most time-consuming part of the process which was also seen in Chapter 4.

![Figure 18. Times before the test started after the test was created.](image)

As we can see from Figure 18, a minimum waiting time for a test run was 23 minutes and the average was around 93.3 minutes. That means a tester will wait for test results over one and a half hour more likely than a half hour. From those numbers, we can make a conclusion that executing tests is not so fast. Waiting for a suitable test environment from the cloud takes the longest time. Locally test environment is always available beside a tester. However, it must be remembered that locally a tester needs to update all software to a test environment manually before starting to run tests. However, the web application seemed to be reliable because all tests were executed at some point.
7. Fault Detection with Partly-Simulated SUT

In this chapter, the aim is to find empirical data about a fault detection with partly-simulated SUT. The process will be described and results summarised.

7.1 Fault Detection

As previous chapters gave results to Requirements 1, Requirements 2 and Requirements 3, this chapter gives results to Requirements 4. Let us remember that those requirements were “Fault should be detected with partly-simulated SUT” and “Simulator’s software shouldn’t cause new faults”. From those requirements, we can summarise the research question for this chapter which is following.

Sub-RQ6: How high is the fault detection capability of the partly-simulated SUT against the real SUT?

Sub-RQ6 is the crucial research question because a fault detection with a partly-simulated SUT is one of the key concerns in the cloud service. This question came up also in the literature review where Mårtensson et al. (2016) state that simulation solutions in embedded systems have fault-detection concerns. If partly-simulated SUTs cannot find faults, then all tests are pretty much useless because a fault will not be detected during a testing and there is a chance that customers find the fault. Also, we succeeded to automate data calls in the cloud but we don’t know how simulated calls can find faults. If we manage to verify that simulated calls find same faults than a real SUT, it defines the cloud’s usefulness for the testing.

Also, besides concerns of fault detection, the interviews told us that the target team has concerns about the simulator software possibly causing new faults. If the simulator’s software causes new faults, then testers need to test more and there might be disagreement with software developers about which fault is simulator’s fault and which one is the software fault.

7.2 Data Against Requirements 4

This chapter provides answers against Requirements 4. Table 9 sum up answers which are explained later in this chapter.

Table 9. Results of Requirements 4.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault should be detected with partly-simulated SUT</td>
<td>Fail</td>
</tr>
<tr>
<td>Simulator’s software shouldn’t cause new faults</td>
<td>Fail</td>
</tr>
</tbody>
</table>

7.2.1 Fault Detection of Official Test Runs

We were able to analyse 34 official test results which were executed in the cloud. Those tests were previously implemented tests “Base station reset”, “Cells block and unblock” and “Base station block and unblock”. The following table shows how many of those test
executions failed because of a software fault and the table also shows how many executions passed.

![Official Test Runs](image)

**Figure 19.** Summary of passed and failed tests.

Overall 34 test results were analysed and Figure 19 shows results from those tests. Tests were executed with different software builds. We can see from Figure 19 that 26 tests passed and 8 tests failed. From these statistics, we can conclude that partly-simulated SUT can find faults because tests also failed which is always a good thing because aim for testing is to find faults.

<table>
<thead>
<tr>
<th>Test Description</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1 (Cells block and unblock)</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Test 2 (Base station block and unblock)</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Test 3 (Base station reset)</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 10. Test-specific statistics.

As Table 10 shows, there were not much difference about fault detection capability between different tests. Test 1 was executed the largest number of times, that is, 12 times but it failed three times like Test 2 which was executed 10 times. Test 3 was executed 9 times and it failed 2 times.

<table>
<thead>
<tr>
<th>Test 1 faults:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x Software faults</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test 2 faults:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x Simulator fault</td>
</tr>
<tr>
<td>2x Software faults</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test 3 faults:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x Software faults</td>
</tr>
</tbody>
</table>
As Table 11 shows, most failures were caused by the tested software. However, there was one test which failed because of the fault in the simulator's software. From these statistics, we can conclude that simulator’s software causes new problems.

7.2.2 Specific Faults Under Investigation

Fault 1: RRU Specific Faults

One of the most common answers when executing interviews was that RRU specific faults cannot be detected. Based on that we can start to collect relevant data for answering to the concern. In this case, the best way was to use some known fault in some specific software build from a history. An internal database was used for selecting the fault because the database contains all fault reports from history.

The first point was to choose a right fault report which contains all information on how the fault was produced and which hardware configuration was used. With the information, the fault can be reproduced by using same software build and RRU which caused the fault. The most important criteria when selecting a fault was the fact that the fault must come from the RRU specific software component. It must be checked that a correction for the fault was successful when the modification was done to the RRU software component which is a part of the tested software build. The fault situation must be reproduced by imitating the fault situation from the report.

A good deal of interesting reports which contained a problem when RRU was configured during the base station start-up were found. That kind of an alarm indicates that RRU is out of order and a cell phone cannot start data transfer for the affected cells for which configuration was failed. That kind of an alarm might not raise when RRU is simulated which leads to the situation where the fault is not detected. However, corrections of some faults did not come from RRU software component so those faults were not suitable because the goal was to find a fault that comes from RRU software component.

Finally, a suitable fault report for this purpose was found. When starting to reserve a test line by using a cloud tool there was a problem because the software build in the fault report was so old that it couldn’t be included in the reservation. A test line needed to be reserved with newer software build and update the old software manually to the SUT.

The base station start-up was tested separately with a real base station configuration and a partly-simulated base station configuration. The configuration type was identical in both cases. The configuration included only one cell with one RRU and BBU. The fault was permanent which means that it was visible all the time when the base station was powered on. Also, it was easy to produce manually with the base station controlling tool.

As Figure 20 and Figure 21 display, when the start-up was executed in both environments it became clear that alarm did not raise with simulated RRU software. Because the fault was permanent and visible all the time in the real SUT, we can say that all faults will not be detected with partly-simulated SUT.

The conclusion is that simulated hardware does not actually use RRU software component. This kind of faults will not be found because simulated RRU acts an ideal way instead of a real way. It is a clear disadvantage because RRU software component is a part of the software under testing so those faults should be detected.
Figure 20. Critical alarm visible in the screenshot taken from BTS controlling tool when using the real RRU.

Figure 21. Critical alarm not visible in the screenshot taken from BTS controlling tool when using the simulated RRU.

Fault 2: Call Problem after DSP Crashed

Now we have the result that some faults cannot be detected when using simulated RRUs, which is good to know. However, there is still a lot of different faults that come from different software components. One tester from the target team gave an idea that we could check if the partly-simulated SUT can find a certain data call fault.

DSP crash is a common fault that can happen at least when the base station is started. It raises a certain alarm. If all DSP crashes, recovery reset to BBU comes. This kind of a recovery action supports the literature where it was stated that output can save the system from failure if software functionality breaks (Karmore & Mahajan, 2016). After recovery reset all cells are on air again and data calls should be possible. However, a tester said that there is a fault in one software build where calls are not possible.

That fault was tested with partly-simulated SUT and the fault was detected which was a good finding because it gave a positive feedback not only that the partly-simulated SUT can find some important faults but also that executing simulated call can find faults.

As Figure 22 shows, cells 11, 12, 13 were green which indicates on air state but terminal window states that ue_attach command failed. That means UE like a cell phone cannot attach to a cell even though the cells are on air state. Also, log files show that ATA value is over 1000 when it should be 0 or 1.
Figure 22. Screenshot taken from BTS controlling tool from a cloud environment.

Other Faults from the Research

Even though the fault that came from RRU software component was not found with simulated RRU's, there were many faults that were detected in different situations during the iteration process of tests development. Many unexpected alarms such as DSP crashes were detected with different software builds which caused test failures.

Table 12. Fault detection statistics.

<table>
<thead>
<tr>
<th>Fault Description</th>
<th>Real SUT</th>
<th>Partly-simulated SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRU software component failure</td>
<td>Detected</td>
<td>Not detected</td>
</tr>
<tr>
<td>Data call failure 1</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Cells configuration failure</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Unexpected crash</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Unexpected alarm 1</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Unexpected alarm 2</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>RRUs disappears after detection</td>
<td>N/A</td>
<td>Detected</td>
</tr>
<tr>
<td>Data call failure 2</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>RRUs undetected after start-up</td>
<td>N/A</td>
<td>Detected</td>
</tr>
</tbody>
</table>

Table 12 shows that there are at least two new faults that the simulator’s software caused. That kind of faults cannot be found with the real SUT because it doesn’t use simulator’s software. Faults in the simulator’s software were not seen as a good thing in the target team because it increases testers’ tasks and testers need to focus on irrelevant issues. From those statistics, we can also verify that partly-simulated SUT causes new faults so that concern from interviews is real.

We should remember literature which stated that simulation-based cloud testing should be investigated more (Mårtensson et al., 2016; Tao & Gao, 2016; Riungu-Kalliosaari et al., 2016). Well, here we have some statistics from the industrial level which show that there is a chance that all faults won’t be detected when simulating the SUT, which is a clear disadvantage.
However, there are six different faults that were visible in both kinds of environments so fault detection is not as bad as was expected. Only one of these faults was not found with partly-simulated SUT.
8. Discussion

This chapter will provide the summary of the research results. Also, those results will be discussed against the findings from the literature review. This research has also limitations which will be explained in this section. Furthermore, suggestions for further study are discussed.

8.1 Summary

Now it is time to summarise results of the research against the literature as the last Sub-RQ7 indicates. As Riungu-Kalliosaari et al. (2016) states that more industrial level research about cloud-based testing is needed, this research brings some simple knowledge to that scope.

Sub-RQ7: How do the results of cloud-based testing compare against information in the scientific field?

Findings from the research process indicate that besides benefits, cloud-based testing will bring some restrictions to the target team’s testing area. What literature also pointed out was that cloud-based testing brings on-demand testing with elastic resources such as test environments (Tao & Gao, 2016). According to the results, we can confirm that because in the cloud, testers does not need to bother anymore building or maintaining test environments so Incki, Ari & Sözer (2012) are right when stating that testing in the cloud reduces costs because of installing and maintaining a test environment’s software doesn’t have to be manual work. For example, in the cloud, there is no need to update manually any software to a test environment or change heavy hardware units to the SUT. The cloud has more test environments which were already initialized when testers received those environments.

However, this research confirms that the cloud does not bring only benefits because there are clearly some restrictions. Results show that waiting times in the cloud can be long and that is the reason why the cloud might not be so suitable for tests which need to be executed immediately. Especially continuous integration testing requires that all software builds should be tested with 100 % pass rate (Mårtensson et al., 2016). When many software builds come in a day and cloud-based tests can have waiting times of many hours, dozens of tests are difficult to conduct with all test environments. Even though the web application would work nicely like in this study, it doesn’t matter if tests would not be executed quickly.

There are no tests in the cloud if tests are not developed with some test automation tool. As the literature pointed out, test automation tool can be harmful if it doesn’t support the requirements that testing has (Garousi & Mäntylä, 2016). Well, in this case, we can say that test automation tool of the cloud is useful for the target team even though it doesn’t offer functionalities to create as comprehensive tests as currently. The reason why the tool still can be useful is that the tool offers a possibility to automate some manual tests which are not currently automated like those data transfer tests. Also, tests’ maintainability seemed to be good. Also, the tool included some problems like recyclability problem when using other team’s test resources was not as easy as was expected.
A fault detection with the cloud’s partly-simulated SUTs was the crucial part of this research which was also pointed out in the literature (Mårtensson et al., 2016). Most faults were found but there was one fault which was not found. It was RRU specific fault that was visible with a certain type of RRU. Those faults which come from RRU software component are less common than faults which come from other software components but it is still a part of the tested software so it is a clear disadvantage that the RRU specific fault was not found. Also, two faults were found which the simulator’s software caused. That is also a disadvantage according to the target team’s testers because new faults mean that there is extra software which must be tested.

However, simulated data call tests found same faults that the real SUT find which says that the manual test which was automated is successful because it can find same faults automatically that are currently executed manually. Besides studying some specific faults this research involved 34 official test runs where test passed in 26 times, failed in seven times. From those statistics, we can only say that the partly-simulated SUT can find some faults.

As Table 13 presents, we can say that the cloud service will not properly fulfil all testing demands in the target team because overall eight requirements did not work and eight requirements worked. There were also four requirements where we have not enough data to say if those work or not. However, besides the current test automation, the cloud tests can bring benefits in. Basic tests, which include data calls, can be executed easily for many different SUTs when currently there are only about ten SUTs in the test automation. Simulated hardware makes it possible to simulate different types of hardware that are currently in use.

Table 13. Statistics of Requirements.

<table>
<thead>
<tr>
<th>Results</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed</td>
<td>8</td>
</tr>
<tr>
<td>Failed</td>
<td>8</td>
</tr>
<tr>
<td>N/A</td>
<td>4</td>
</tr>
</tbody>
</table>

8.2 Limitations and Future Research

In this research, more empirical data to all research questions could have been collected so that the research’s validity could have been better (Easterbrook et al., 2008). Also, some results are based on the researcher’s observations, and it can be discussed whether it is such a good way to execute the research. However, the target team was satisfied because they have not used the cloud-based testing and the target service before. This research gave a lot of information and courage to them for investigate the cloud-based testing service more and develop automated tests further in the cloud.

The requirement which state that more hardware configurations should be able to test was not studied enough. That requirement should be studied further because that was one of the most important requirement and that is the big reason why the simulation-based cloud solution is developed in the company.

The literature points out that all manual tests can’t be automated in the embedded software development because those might need human attention which test automation might
never offer. Possibility to automate current manual tests could have been investigated more. In this study, only one manual test was automated but the higher management of the case company are pushing the target team to reduce manual tests, for example, by automating those. The target team have already reduced manual tests because there is not enough time to test those due to shorter delivery cycles.

Dropping off tests can be a problem because some faults might not be detected before the customer gets the product. Dropping off tests seems to argue against the fact that testing is the only way to make the product 100% flawless. Of course, it is understandable when software development is implementing continuous integration more. Then there isn’t time to execute manual tests so much but still, it raises the question whether embedded software testing is losing its value in companies’ eyes. (Karmore & Mahajan, 2016; Mårtensson et al., 2016)

When conducting the research, it was not considered how much effort it takes to manage a test environment in the cloud which is a relevant question according to the target team’s testers. It should be remembered that always it is someone’s duty to maintain test environments if automatic initialization won’t work. For example, if the hardware is broken, someone must replace it. That aspect was not considered. Also, in this section, it would be good to highlight that cloud-based testing security issues should be investigated more as the literature pointed out (Riungu-Kalliosaari et al., 2016).
References


Appendix A. The Interview Structure

Päiväys ja aika:

Haastattelija:

Esittely:
Haluaisin kerätä listaa mahdollisista hyödyistä ja huolista Robot Frameworkilla toteutetusta testiautomatiosta ja testiypäristöistä pilvessä.
On tärkeätä kuulla teidän vaatimuksen, koska teille tätä työtä tehdään
Tämä haastattelu kestää noin 15 min
Saa ja pitää vastata vapaasti, mitä mieleen tulee.

1. Haastateltavan tiedot:

   Rooli (Jos muu kuin testaus):
   Ikä:

1. Listaa tähän huollet
   - Testiypäristöt pilvessä?

   - Testausautomaatio työkalu?

   - Muut asiat?

2. Listaa tähän hyödyt.

   - Testiypäristöt pilvessä?

   - Testausautomaatio työkalu?

   - Muut asiat?
## Appendix B. Waiting Times Before Test Starts

<table>
<thead>
<tr>
<th>Test Environment</th>
<th>Test Creation Time</th>
<th>Environment Waiting Time in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Line B</td>
<td>7:35</td>
<td>35</td>
</tr>
<tr>
<td>Test Line B</td>
<td>8:55</td>
<td>70</td>
</tr>
<tr>
<td>Test Line B</td>
<td>15:11</td>
<td>37</td>
</tr>
<tr>
<td>Test Line B</td>
<td>15:12</td>
<td>74</td>
</tr>
<tr>
<td>Test Line B</td>
<td>15:13</td>
<td>69</td>
</tr>
<tr>
<td>Test Line A</td>
<td>7:56</td>
<td>231</td>
</tr>
<tr>
<td>Test Line A</td>
<td>7:58</td>
<td>52</td>
</tr>
<tr>
<td>Test Line C</td>
<td>13:41</td>
<td>78</td>
</tr>
<tr>
<td>Test Line B</td>
<td>7:38</td>
<td>23</td>
</tr>
<tr>
<td>Test Line B</td>
<td>9:25</td>
<td>44</td>
</tr>
<tr>
<td>Test Line B</td>
<td>11:36</td>
<td>91</td>
</tr>
<tr>
<td>Test Line B</td>
<td>9:02</td>
<td>28</td>
</tr>
<tr>
<td>Test Line B</td>
<td>9:02</td>
<td>28</td>
</tr>
<tr>
<td>Test Line B</td>
<td>11:40</td>
<td>86</td>
</tr>
<tr>
<td>Test Line B</td>
<td>11:40</td>
<td>68</td>
</tr>
<tr>
<td>Test Line B</td>
<td>11:40</td>
<td>90</td>
</tr>
<tr>
<td>Test Line C</td>
<td>14:30</td>
<td>77</td>
</tr>
<tr>
<td>Test Line C</td>
<td>14:30</td>
<td>184</td>
</tr>
<tr>
<td>Test Line C</td>
<td>14:30</td>
<td>129</td>
</tr>
<tr>
<td>Test Line C</td>
<td>11:05</td>
<td>45</td>
</tr>
<tr>
<td>Test Line C</td>
<td>11:05</td>
<td>100</td>
</tr>
<tr>
<td>Test Line C</td>
<td>11:06</td>
<td>103</td>
</tr>
<tr>
<td>Test Line C</td>
<td>13:48</td>
<td>178</td>
</tr>
<tr>
<td>Test Line C</td>
<td>13:48</td>
<td>230</td>
</tr>
<tr>
<td>Test Line C</td>
<td>13:48</td>
<td>242</td>
</tr>
<tr>
<td>Test Line C</td>
<td>8:22</td>
<td>34</td>
</tr>
</tbody>
</table>
### Appendix C. Results of All Requirements

<table>
<thead>
<tr>
<th>Requirements 1: Test environments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test environment reservations should be reliable</td>
<td>Fail</td>
</tr>
<tr>
<td>Reservation waiting times shouldn't be too high</td>
<td>Fail</td>
</tr>
<tr>
<td>Partly-simulated SUTs should support more configurations</td>
<td>N/A</td>
</tr>
<tr>
<td>SUT should be revived effectively after it crashes</td>
<td>N/A</td>
</tr>
<tr>
<td>Test environments should be initialized automatically</td>
<td>Pass</td>
</tr>
<tr>
<td>Should be able to simulate different RRU types</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirements 2: Test automation development</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test scripts maintainability should be good</td>
<td>Pass</td>
</tr>
<tr>
<td>Other teams test resources should be accessible</td>
<td>Fail</td>
</tr>
<tr>
<td>Changing hardware configuration for simulator should be easy</td>
<td>Pass</td>
</tr>
<tr>
<td>Should be possible to develop comprehensive tests</td>
<td>Fail</td>
</tr>
<tr>
<td>Should be possible to automate current manual tests</td>
<td>Pass</td>
</tr>
<tr>
<td>Tool should have good documentation and instructions</td>
<td>Fail</td>
</tr>
<tr>
<td>Automation tool should work with other testing tools</td>
<td>Pass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirements 3: Test executions through the web application</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>The web application should be reliable</td>
<td>Pass</td>
</tr>
<tr>
<td>Learning to use the web application should be easy</td>
<td>Pass</td>
</tr>
<tr>
<td>Testing or retesting should be easy</td>
<td>Fail</td>
</tr>
<tr>
<td>The web application should be accessible everywhere</td>
<td>N/A</td>
</tr>
<tr>
<td>The web application should work with the reporting tool</td>
<td>Pass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirements 4: Partly-simulated SUT</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault should be detected with partly-simulated SUT</td>
<td>Fail</td>
</tr>
<tr>
<td>Simulator's software shouldn't cause new faults</td>
<td>Fail</td>
</tr>
</tbody>
</table>