CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The effect of test case design in software testing bots

Bachelor of Science Thesis in Software Engineering and Management

Martin Chukaleski
Samer Daknache

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Test case design for software testing bots
Outlining some of the current test design challenges associated with system testing bots and discussing possible mitigation strategies

Supervisors: FRANCISCO GOMES DE OLIVEIRA NETO and LINDA ERLENHOV
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

The effect of test case design in software testing bots

Martin Chukaleski
Department of Computer Science
and Engineering
University of Gothenburg
Gothenburg, Sweden
guschuma@student.gu.se

Samer Daknache
Department of Computer Science
and Engineering
University of Gothenburg
Gothenburg, Sweden
samer@daknache.se

***Abstract*—Traditional approaches of testing in software development include running the test cases on a software component, referred to as unit testing, which usually only tests a specific part of a component, as opposed to testing the whole flow of the system (end-to-end testing). Test bots are software automation tools that help improve the system testing via automation, which is beneficial for development teams as the test bots help decrease the amount of time spent on testing. As development projects become larger, it is important to focus on improving the test bot's effectiveness. The test bots run a set of test cases that check whether the system under test meets the requirements set forth by the customer. This thesis uses a case study approach to investigate how test case designs can affect the test bots, and by using the findings gathered from the study, we aim to create a guide for test design schema for such bots. Furthermore, this study aims to find how the software testing practices in an IT company can differ from what the literature presents. We identify the main challenges when using test bots in the automotive industry and a guideline is composed of seven steps to aid stakeholders in designing tests where test bots are part of the testing cycles.**

***Index Terms*—test bots, software testing, system testing, end-to-end testing, test results, test case design, automotive industry.**

I. Introduction

Testing is amongst the most popular techniques used to perform quality assurance on software systems. Even though there are many types of testing techniques, in practice, one of the most expensive and time-consuming techniques is functional testing [1]. Functionality of a system can be stated as the external behavior that satisfies all user requirements. There are multiple types of functional testing methods which could be performed at different levels of testing, such as: *unit testing*, *integration testing* and *system testing*. Functional black box testing, which can also be stated as behavioral testing, is a software testing method in which the internal structure, design and implementation of the system is being tested [2]. A test case takes a set of input (also referred to as *test data*), which is used to perform some specific computations resulting in some data output. The output is compared to the expected result, thus stating whether the test passes (if both match) or fails. End-to-end (E2E) testing is a type of testing used to evaluate if the flow of an application is behaving as designed from beginning to the end.

The current literature provides us with knowledge regarding the test case creation process [3], how to create end-to-end or system tests [1] and how to avoid test design smells (poorly designed tests) [2]. End-to-end testing can be automated by making use of software automation tools, referred to as *test bots*, which help decrease the time development teams spend on software testing. Test bots are part of a wider range of software bots, particularly DevBots [4], which can be described as software applications that have a degree of autonomy that can execute automated tasks in a repetitive schedule [5]. Test bots differ from software bots as these types of testing bots are different than the chat bots, voice bots or gaming bots as they do not mimic the human social interaction. Test bots fall in the group of productivity bots, because they improve the development team's productivity by automating the execution of testing tasks [5].

Current literature however lacks studies focusing on how to design system tests which are carried out by software bots. The goals of this study are therefore to:

- Discover current industry practices when it comes to end-to-end system test case design with the help of software bots, which in a way act as the executor of the tests.
  Our expected contribution for this goal is to depict the challenges associated with the design of system tests for test bots.
- Identify the difference between good and bad test case design for system test and describe the underlying factors.
  Our expected contribution are guidelines on how to design system test for test bots.

A case study was performed in a company within the software engineering industry using an investigative approach where practitioners were interviewed and software artefacts related to test bots were analyzed. The data collected from this study along with the analysis, was cross-evaluated and gave us information that guided us into creating a test design schema for test bots. The motivation behind why this research was performed is due to almost 50 to 70% of effort from the total software development is spent on testing and approximately 50 to 70% of it lies in the group of functionality testing [1]. Establishing a good test design schema for the automated test

bots can potentially help provide better test coverage, which in turn can help increase test effectiveness [6]. The usage of the term test effectiveness implies that the test case should be simple, faster when executing and lastly less complex (e.g., in terms of cyclomatic complexity and lines of code). It is important to mention that this study does not aim to pursue *defect detection rate*.

A. Statement of the problem

Though there are industry standards for test design techniques [7], such standards are not completely adopted by development teams. This can lead to teams incorporating different testing designs thus the test case design affects the test bots' performance differently. In other words, teams do not adopt a standard when designing tests and therefore our hypothesis is that such differences can affect the effectiveness of the test bot. Without a set of guidelines, developers may end up creating test bots that are less time efficient and potentially more complex than needed. Researchers interested in testing automation have the potential to benefit from this study as it covers test bots' role of automating test execution and reporting back the test results.

B. Research questions

These are the research questions (RQ's) that we answer with our case study:

1) RQ1: What are some of the challenges and practices when designing and executing system tests on software bots?
2) RQ2: To what extent does the test design affect the effectiveness of the test bot?
3) RQ2.1: What are the differences between good and bad test case design for system testing when using test bots?
4) RQ2.2: How to design good test cases for system test executed on a test bot?

C. Structure of the thesis

Section II of this paper outlines the literature collected from the problem domain and how it is related to our research. Section III details our methodology, including data collection and evaluation methods, as well as the scope of our thesis. The limitations that arise in this research are also outlined in this section. Section IV contains the results from this study, depicting the functionality of the system, including the software build pipeline, testing process, test bot description and test case design. The answers to our research questions are discussed in Section V. Finally in Section VI we discuss the conclusions of this research and its relevance to future studies.

II. Related work

A. E2E system testing and test case design

Tsai et al. [1] gives an overview of how to design E2E tests, and depicts the creation of test scenario specification, test case generation and tool support. The focus is specifically on the test scenarios and test case generation. As defined in this study the aforementioned scenarios represent a single functionality from the perspective of the user's point of view. The authors also analyze the relationship between different functionality tests, the conditions used and the consistency between them and they also reveal some ratios which illustrate the effort or time spent when creating tests for the system and the portion which is taken by the integration tests.

Elssamadisy and Whitmore [2] state the current functional testing formats, which are derived from a combination of the author's experiences with functional testing in several agile development projects. In the study it is mentioned that functional testing can become more costly than its benefits if it is not implemented correctly. These problems can be found in the test implementation practices or in the architecture of the system under test.

Based on Hooda and Chhillar [3] we can gain deeper knowledge on the various phases that a test needs to go through to be designed, and those are: test analysis, test planning, test case data preparation, test execution, bug logging and tracking and closure. The authors also describe how developers can ensure the quality of various types of software applications by performing certain types of testing techniques and optimized software testing processes.

Mockus et al. [6] investigate how test coverage affects test effectiveness and the relationship between test effort and the level of test coverage. The common theme between our research and the aforementioned study is the examination of test effectiveness along with test coverage.

Oladimeji et al. [8] outlines in their paper the different levels of testing. One of the levels of testing that the authors cover is system testing, which is a common theme shared with this paper.

Laventhal et al. [9] discuss in their paper the relevance of negative and positive tests and how testers are exhibiting positive test bias, which can affect the quality of testing, which is relevant within our study of test case design.

B. DevBots

Erlenhov et al.'s [4] study proposes a face-based taxonomy of existing DevBots while also providing definition and vision of future DevBots. This is relevant within our study of DevBots at it helps gain a better insight of how different bots are classified.

According to Lebeuf et al. [5] software bots can help improve the efficiency of every phase of the software development life cycle, including test coding. The paper outlines the different types of bots and how they can respectively help improve software development. While the paper is beneficial for outlining the difference between bots, it does not dive deeper into the different bots, but rather provides an overview of how these bots can be beneficial. One common theme that the paper covers, which our study includes, is the test bots and how they can be used to the developer's advantage. The authors also described how software bots are currently used in the software engineering industry and the bots are classified in

TABLE I
CASE STUDY PLANNING ACCORDING TO GUIDELINES BY RUNESON ET AL. [10]

Objective	Exploration
The context	Black box, end-to-end system testing
The cases	One project from the automotive software industry
Theory	Test case design, DevBots
Research questions	RQ1, RQ2, RQ2.1 & RQ2.2
Methods	Direct and independent data collection methods
Selection strategy	Project using test bots for system testing
Unit of analysis 1	Qualitative assessment of interviews
Unit of analysis 2	Quantitative assessment of software artefacts

different categories based on the tasks they are used to solve or fulfil.

III. RESEARCH METHODOLOGY

This case study investigates some of the challenges associated with test bots when performing system testing and also proposes guidelines on how to design adequate test cases by performing interviews with practitioners and analyzing software artefacts. The summary of our case study planning is depicted in Table I. The different elements of the case study will be detailed in the upcoming subsections.

A. Case study company description

The company that we chose to perform the case study at is a relatively mature company, located in three offices in three different countries. The company provides Services as a Product (SaaP) for the automotive industry. There are multiple programs in the organization which work for different car manufacturing companies. Our research is going to be performed only within the scope of one of those programs, which is responsible for developing scalable software solutions for a specific car manufacturer. Four interviews were performed within the organization with developers of different expertise: one software architect, two senior and one junior developer.

B. Environments

The case study company uses Amazon Web Services (AWS) and Microsoft Azure as their cloud service providers. A cloud provider is a company that delivers cloud computing based services by providing rented and provider-managed virtual hardware, software, infrastructure and other related services used to store data and host server applications. A software deployment *environment* or tier can be described as computer system or a multitude of those systems in which a software component is deployed and executed. A virtual private cloud (VPC) is a logical division of a service provider's public cloud multi-tenant architecture to support private cloud computing. With this model, companies can achieve the benefits of private cloud, such as a granular control over virtual networks and an isolated environment for sensitive workloads. Workloads can be stated as sensitive, when handling data which is customer private and should not be exposed publicly.

It is worthy to mention that different programs within the organizations have deployed their environments within different VPCs, which elevates the level of service isolation. The service isolation is important because it segregates services that are *Customer* specific within the organization, for example different programs which have different car manufacturer companies as customers have unique VPCs, which allow them to communicate within their scope and prohibit them from leaking delicate information to other systems.

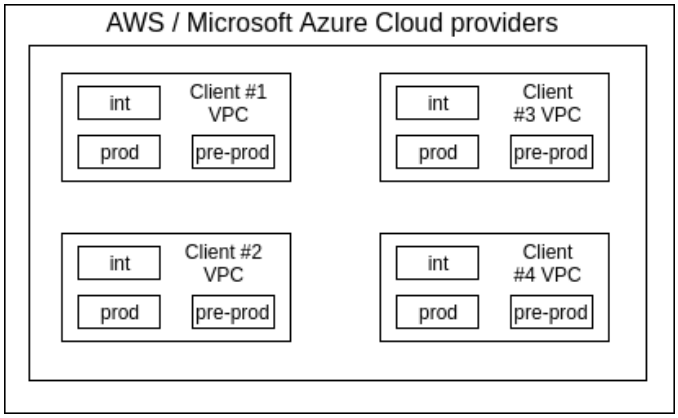

Fig. 1. Environment isolation

The environments which are currently used in the program are: integration (`int`), pre-production (`pre-prod`) and production (`prod`). All of the previously mentioned environments are placed in the same virtual private cloud however, they are isolated and can not share data or communicate between each other. `Int` is used by the developers to continuously deploy and test new features which are created during the time span of one sprint since the team uses SCRUM, with one sprint usually lasting two weeks. However, sometimes sprint duration can be accustomed towards the customer needs. `Pre-prod` is used to deploy only pre-tested versions of features from the int environment which do not contain any faults or bugs as acknowledged by the test suite. In the `pre-prod` environment there is a new code deploy after every sprint has ended, and it is mostly used by the customers to manually test out the functionality and approve its acceptability. `Prod` is used by businesses or private individuals around the world, accepted code from `pre-prod` is deployed to `prod` only when the customer demands an upgrade. It is worthy to note that the same test suites are always used in all three environments. Figure 1 depicts the isolation between the different environments which are hosted under one VPC on the Amazon cloud service.

C. Data collection

We performed initially a semi-structured interview with developers which were familiar with the overall scope of software test bots and have worked on their development. The interview method was chosen because it allowed us to engage with participants directly and obtain contextual, nuanced and authentic answers [11]. During the interview, questions can always be additionally clarified and re-formulated, thus eliminating duality and inconsistency [12]. However, interviews

also have their disadvantages, as they can sometimes be potentially intrusive for the participant, time-consuming, leading to incomplete answers, and they also can be prone to bias [12]. In order to mitigate some of these disadvantages, it was decided that the interview would be time boxed to approximately one hour and the participants would be informed of their anonymity. Moreover, the interviews were recorded only if the participants agreed. The predefined questions were identical for all interviewees in order to collect consistent data. Because the interviews were semi-structured, deviations were expected, thus new sub-questions were formulated during some of the interviews.

The interviewees were selected based on having previous experience with the test bots. It is important to mention that the authors were employees in the case company working in one of their projects as developers and that the participants were their colleagues. Moreover, the test research was orthogonal to the work activities performed by the researchers. Additionally, we also collected data from software artefacts which included test bot code, test case code, functional requirements (only within the context of end-to-end testing).

To get a better understanding of the testing process, an observation was done within the program team during the development of the test cases together with the test bot code. During the observation, the authors witnessed the processes used when designing the test cases and also the information which were analyzed by the developers. According to Smith [13] observations can be good sources for providing additional information about a particular group. They can be used to generate qualitative data and quantitative data: frequency counts, mean length of interactions, and instructional time. One of the biggest advantages of conducting observations is that it allowed us to observe what people actually do or say, rather than what they say they would do. Observations can be made in real situations, thus giving us access to the context and meaning surrounding what people say and do. Since all of the data was within the ownership of the organization, the information related was anonymized.

D. Data analysis

After the data was collected, the analysis was performed in several phases. The first stage was transcribing every voice recording which was made during the interviews. The second stage was organizing our data in a way that it would be easier to read and understand or map between the different participants. This was done by grouping different questions or sub-questions from the interviews into categories that actually were a part of the same context, consequently all of the answers to those questions were grouped accordingly. The third phase was coding our data, this can be described as compressing the information into easily understandable concepts for a more efficient analysis process. Codes can be derived from theories or relevant research findings or from the research objectives.

The data analysis method used was thematic analysis, which falls within the group of qualitative analysis methods, which is used to find patterns in the raw data and uses those similarities as the base for the coding [14]. With the help of this type of analysis, we generated categories which summarized the data gathered and expressed key themes and processes.

The categories which were created from our coding contained three key features: category label, category description and text [15]. The most influential approach that we could find to perform the thematic analysis was by following Braun and Clarke's [16] six step framework. The steps were executed in a linear path, one after another. However when dealing with more complex data they can be re-iterated forward and backward many times, until the required codes and themes are generated.

1. Become familiar with the data: In the first step we reviewed and read all the interview transcripts multiple times, to get familiar with the whole body of the data.

2. Generate initial codes: In the second phase we organized and coded our data in a systematic way, thus reducing the data in small meaningful pieces. Since we focused on specific research questions, we chose to select the theoretical thematic analysis rather than an inductive one. Every segment of the collected data that was found significant or that captured something interesting about our research question was coded. We did not code every single line or word from the transcripts. We used open coding, meaning that we did not adopt any predefined codes, rather the codes were generated and modified during the coding process.

3. Search for themes: In the third step we examined the newly generated codes and grouped them according to their shared context into various themes, where the codes were organized into broader themes that have unique connotations about the research questions.

4. Review themes: During the fourth step, the review phase, the themes were analyzed if they were coherent, when established otherwise, modifications to the themes were made. After their creation, the data that fell within the context of a theme was grouped accordingly.

5. Define themes: In the fifth stage, the final refinement of the themes was achieved. The goal here was to discover what were the themes stating, were there any sub themes, if there were some, how do they relate to each other and the main theme.

6. Write-up: In the final step the outcome was the actual written report, together with the presentation of themes, codes and associations found within the data set. The tool of choice used for the data analysis was Microsoft Excel.

Lastly, our findings which were created from the data that was transcribed, organized and coded, were additionally validated by the participants of the interview. This was performed so that we could diminish any misunderstanding from our side towards the program in the organization. The software artefacts (*test bot* and *test case code*) which were collected were used to get a better understanding of the design of the test bots and their test cases, which was also used to generate insights and inspiration for the optimal test design that we were created. For the two research questions different data sources were analyzed and used to generate their respective answers. Only

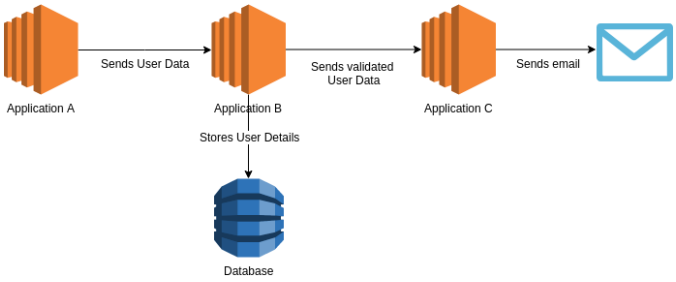

Fig. 2. Microservices

interview data was used for RQ1, on the other hand for RQ2 a variety of data sources were utilized: software artefacts (test bot and test suite code), functional requirements and the testing process (from interview data).

IV. Results

A. System under test

This section contains the data collected regarding the relevant elements in using test bots in the case company, including how the test bots are instrumented, and how they are used.

System overview: The system under test (SUT) exercised by the test bot can be described as a microservice based application, where multiple services communicate with each other with the help of Representational State Transfer (REST) endpoints and/or Data streams. A REST-full web service is represented by a Universal Resource Locator (URL) which when invoked can be used to retrieve, create, update or remove a specific resource or a collection of resources in the system which it interacts with. Data streaming is the process of sending bytes of data continuously rather than in batches, It is most often used for sending or receiving small sized data in a continuous flow as the data is created.

System functionality: The system under test can have multiple functionalities which are required by the customer. If we take for example a service which has the responsibility to create a new user, that singular function can be composed of several microservice based components, which in turn describes that for one request to be fulfilled we have to use three components A, B and C. Application A is the actual website which has placeholders for the user details, once it retrieves them it sends the data to application B, than the data is first verified if it is valid and afterwards stored in a relational database. Finally this chain is completed by application C, which is in charge of sending a confirmation email to the designated user email address. From this, we can conclude that when testing a specific functionality of the system, multiple request will be made to different components which are expected to be available and ready to serve the demanding entity with the correct data output, see Figure 2.

It is also important to note that the system usually interacts with 3rd party car Original Equipment Manufacturer (OEM) back-end systems, which are necessary when gathering data from the vehicles such as fuel level, GPS position and lock/unlocked states.

B. Build pipeline system

The organization has its own in house developed system which is responsible for building the software artefacts that are constantly updated by the developers. It has support for multiple languages like Java, Python, GO, Rust, etc. Once a microservice is updated, an incremental version number is assigned along with the state of the build, where a build state will result in a red or green label being displayed, depending on if the build fails or succeeds, respectively. The component or application can be setup so that when the building process is successful (all test have passed), it can be auto deployed to a predefined environment like `int`, or manually deployed to a specific environment specified by the developers. When a build has failed, the system also displays the error logs, which come in handy for the developers as they outline the fault. The stored code version is very helpful in cases when there is a new build which has faults which are not detected by the test suite, as in those circumstances the development team can roll-back or use an older and more stable version. The build pipeline system has the ability to manage dependencies for the software applications, for example if $Application\ A$ is dependant on $Application\ B$, by stating the explicit build version number of B inside the configuration file of A, which allows the system to fetch the correct artifact from its storage and use it when generating the build for A.

C. Test bot

A test bot has the role of executing a pre-designed test suite, where the test suite can contain test cases that fall within the context of load testing, integration testing, system testing, etc. For the scope of this study we are going to focus only on test bots that are performing system testing. The responsibility of the test bots within the case study company is only to execute the test cases, hence they are not responsible for creating mocked or simulated environments before running the tests, rather the bot is deployed within the environment that it needs to test. Figure 3 depicts the architecture of a test bot and the systems it interacts with. The test bots used in the program have the task of performing end-to-end tests with a specific rate on different functionalities of the system. Depending on the tested functionality, the rate can vary from bot to bot, resulting in a timed schedule for test execution. Usually within the organization higher rates are considered from five to thirty minutes and lower rates fall within the range of one to five hours. Depending on the testing context, whether the system that needs to be tested is back-end or front-end oriented, different programming languages will be used to write the test cases and the test bot. For front-end services developers use JavaScript, while on the other hand for back-end services Java, Python and Scala are used.

Test bot work-flow: One test bot can contain multiple test cases which test different services. Next we illustrate the workflow of the test bot for one functionality. If the test bot needs to evaluate the creation of a new user in the system, several steps need to be performed. The user will be created using soft test

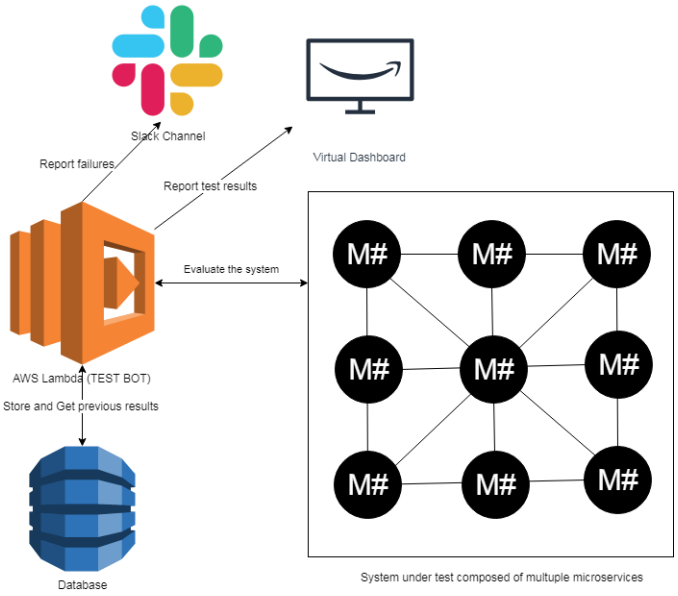

Fig. 3. Test-bot architecture

data, which needs to be used later to verify that a user exists in the database. The term soft test data is used for mocked information, i.e., information that is not real and used only for testing purposes. The scenario for this test case is:

- Create a new user using soft data via the corresponding endpoint
- Send a confirmation email to the email address from the data
- Check if there is new email from the system for the specific user email address
- Check the database if there is a user that exists with the corresponding data

After the execution of this test in a specific isolated environment like (`pre-prod`), depending on the outcome if the user existed or not, the bot will flag the test as a pass or a failure. The second step for the bot is to store the test results together with the time stamps of their execution as a JavaScript Object Notation (JSON) object inside a database. JSON is a lightweight data-interchange format, which can be read and parsed by computer systems faster than other data formats. If the test result is a fail than the error logs will also be a part of the stored object, which can help developers to isolate and discover the faults in the system. Once the logs are stored, the data is published on a virtual dashboard which is used by the whole team to view the state of the system. During the event that the test case fails, a slack-bot notifies the corresponding team about the test suite execution status using the dedicated slack communication channel.

D. Testing process

The testing process depicted in Figure 4 outlines how the program customer requirements are passed on to the product owner who creates user stories, which are short sentences that

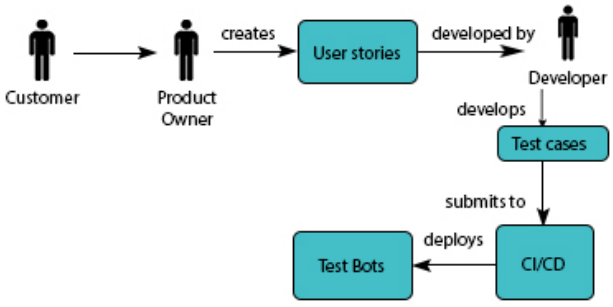

Fig. 4. The testing process

outline the requirements of the system in natural language. These user stories are then developed, usually in a team of seven to ten developers and are then submitted to the internal Git version control system, which then deploys the test bot in the specified environment using the CI/CD (Continuous Integration / Continuous Deployment) build pipeline system.

To design the test cases, the developer is required to have knowledge of the previously mentioned programming languages that are used depending on the system under test (front-end or back-end). When designing the test cases for the test bot, the developer also needs to have an overview of the architecture of the component, in order to be able to understand the flow between the different microservices (see Figure 2). Developers also need to review the source code and user documentation to understand what are the states of the system when the functionality has completed successfully or has failed. If we take the aforementioned example in the test bot work-flow section, the successful state would be established when the user details are validated and the confirmation email has been sent, otherwise this functionality has failed.

V. Test Bot Challenges

Table II shows the themes and labels which were generated from the thematic analysis of the interviews. In this section every label will be interpreted and described accordingly. It is important to mention that the challenges presented can only be found within the domain of system tests executed by a test bot, the reason for this are the specific factors which derive from the nature of the test bot and the environment, such as: i) bot automation, ii) bot execution rate (time interval) and iii) actual system's environment. By the actual system's environment we mean an environment which contains a system that is used both by customers and for the system testing.

C1. Designing context aware test cases: Tests can have dependencies to other tests meaning that a particular test (t_c) can only start executing once test (t_b) has finished successfully. Performing this specific chaining on test cases can become quite complex and timely to achieve using traditional Java testing frameworks like JUnit. Issues can

TABLE II
THEMATIC ANALYSIS OF INTERVIEW DATA ACCORDING TO BRAUN AND CLARKE [16]

Challenges	Frameworks	Test data	Test design
Designing context aware test cases	JUnit	High level requirements	Positive flow testing
Changes in the state of the system	AssertJ	User documentation	Negative flow testing
System clean up	Asynchronous programming		In-house built test runner framework
False positives	Synchronous programming		Modular and easy to read
	Server-less functions		Test reports

arise in the event that a test case fails, thus the following dependent tests will be affected by the previous success rate. There are two possible scenarios in this case; the first scenario is to proceed with the rest of the test executions where the subsequent tests which are dependant upon the previous test will fail while independent tests will pass. In this case even though the developers will get the results from all the tests, it is hard to determine whether the test failed because of some dependency or because that functionality was erroneous. The other scenario on the other hand, is to stop the flow of the subsequent test executions and mark the whole result of the test suite as a failure. The second case will lead to a lack of results, consequently making it challenging for the team to further debug the issues.

C2. Changes in the state of the system: Issues can occur when a test suite has executed partially thus affecting the state of the system or in other words leaving corrupt data within the system. In the context of system testing, corrupt data refers to data which has not been processed or transmitted as expected, thus leading to incomplete data models which can later cause system errors such as null pointer exceptions. For instance, consider the end-to-end test scenario where a new customer needs to confirm an email to get access to their created account; if we assume that the test has failed half-way through the confirmation (e.g., problem in connection or availability), there is the risk of introducing partially created faulty users to the system[1]. To mitigate this problem, the test bot needs to perform roll back techniques in order to remove the invalid data that has been generated by the test suite.

C3. System clean up: In scenarios when all of the test cases have completed successfully, the test bot needs to cover the case of cleaning up after themselves, like previously mentioned. That is clearly one severe risk when using test bots, since they are running on an actual system, where in the event that corrupt data is left behind, the system can potentially be placed in an erroneous state. This clean up would be the equivalent of a tear-down of a test where the system state is restored after executing the test suite. Another risk is the introduction of traceability issues due to low activity in the system. In this case the test bot activity could get mixed up with the customer activity, hence bringing confusion to the developers while viewing the application logs to debug a problem. Another risk posed when using automated test bots that run constantly in a system where the actual number of users *(user activity)* within the system is low, traceability issues can present themselves. This is caused by the large amount of application logs generated by the test bot activity together with a small number of user associated logs, hence making it challenging for the developers when searching the respective application log outcome or recreating the specific problem in the system. For instance, consider a scenario where the test bot is sending 100 requests per second and flooding the system with the test associated logs, thus making it difficult to find other issues that could be user related in the logs.

C4. Flaky tests: Flaky tests or false positives can occur when a test case fails, but in reality there is no fault in the system and the functionality is working correctly. This type of test result can consume a lot of a tester's energy and time when it comes to discovering the issues [17]. In the case study company the test bot is used to evaluate a system which has real time connections with different vehicles, and it requires the vehicles' status to perform the different functionalities. Since the bot's test suite has scenarios where it communicates with a vehicle, in the project a raspberry pi[2] has been used to mimic the functionalities of a car and it has been integrated within the actual system environment. In that particular case, one of the problems is that sometimes this can produce flaky test results (e.g., the raspberry pi can lose the wireless connection or suddenly shutdown), this leads to the test reporting a failed state where test cases are dependant on the vehicle status data, when the system is actually working as expected. The conclusion from this challenge is that when performing automated system testing, the stability of third party incorporated systems within the environment have to be taken into consideration since they can affect the test results. One way to mitigate this problem is to mock the vehicle behaviour on a separate cloud server instance which have 99.9% uptime[3]. However, depending on the scenarios this can be costly and time-consuming to develop.

A. Test case design

In this sub-section we will explain the labels which fall within the themes of test case design.

Positive and negative flow testing: Positive testing is the type of testing that can be performed on the system by providing

[1]Note that the tests are done in the environment which users will access.

[2]https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/

[3]https://aws.amazon.com/ec2/faqs/

valid data as input. This type of test checks if the system behaves as expected with normal inputs, which are done to validate that the application does what it is supposed to do. On the other hand, negative testing is a variant of testing that can be performed on the system by providing invalid data as input. This type of testing checks whether a system behaves as expected with the negative inputs, the motivation for creating these tests is to ensure that the application does not do anything out of the ordinary. As stated by the participants a good test suite design needs to cover positive and negative paths in the application workflow, by skipping the negative paths many issues may not be discovered in the system.

According to Leventhal et al. [9] positive test bias can be described as a behavioral anomaly in which software testers tend to test a part of the software with data which confirms the goal of the functionality (often referred in industry as "happy paths"). However in software testing, such scenarios are counter-productive since it is more effective to test with data designed to disprove the hypothesis. Positive test bias is a critical concern in software testing and it has a large negative effect on the quality of testing. This bias is mostly influenced by the expertise level of the software testers and the completeness of the software specification [9].

Test assertion: HTTP response status codes indicate whether a specific HTTP request has been successfully completed. For the system test cases, two assertions are always used. The first assertion evaluates the availability of the system by checking the HTTP status returned (i.e., status OK or status code 200). The second assertion is the comparison between the expected output and the actual output which tests whether the system behaves as intended. For every test case within the test suite the test results are stored inside an object called *test report*.

Test report: The test report contains the results for every test case which is executed in the test suite. There are two types of failures: first type is a consequence of the environment or the testing itself (e.g., if a REST API endpoint is not available), whereas the second type is a consequence of a false assertion (i.e., the output does not match the expected value). Upon a fail, the test report will also store valuable information (e.g., response headers from the REST endpoint), this combination of information can aid developers in resolving issues associated with system functionality.

Modular and readable test: Test cases that are big in size and do multiple assertions on different test results or test multiple functionalities of the system are perceived as complicated. As stated by one of the interviewees, tests that are too complicated are costly due to the need of refactoring when changing the functionality of the system. In those cases developers can find it difficult to change the existing test or even create new test that would follow the same interface, because it is difficult to understand the existing setup, thus this would lead to more time and effort being spent on developing new test cases. Another issue with large test cases is that it takes quite some time for them to finish executing. There are two bottlenecks in the previous case, one of them is that it takes more time to get the test results and the other is the increased cost that the test suite would generate because of higher execution time. As stated by Nguyen et al. [18], almost 40% of time is spent on test planning, analysis and design, hence the fact that following a more complicated setup to create new test will lead to a huge amount of effort spent by testers and developers, this co-aligns with the statement of one participant where they mentioned "too complicated test cases can scare developers as they require a lot of effort and analysis of the existing design".

High level requirements and user documentation: The development team uses high level requirements gathered by the product owner to generate the test data which is used as input and expected output. System testing is dependant on the high-level design specification in the development process, as any problems that arise during the translation of the requirements and design specification would lead to drastic problems as they propagate downwards to the lower levels of the system testing [19]. In other words, problems with the design documentation could lead to the developers creating inappropriate test data which would then create invalid results from the system testing attempts. User documentation is also used to create test data as mentioned by our interviewee "we design the system from the provided customer/user documentation which informs them on how to use the system, and we can use that kind of information ourselves to construct the tests together with the high level requirements". This also co-aligns with what the current literature presents [20].

Future test bot improvements: The participants also stated what kind of future improvements they would like to add to the test bot functionality, one thing they would like to add to the test result report is the time that was needed for the system to perform each functionality in the test suite. Later, this information can be combined with the history of test results, and patterns can be found which would describe how the user interaction spikes affect the execution speed of the system. This data is valuable and can be used to extensively configure the application for faster performance. Performance testing determines if a system works as fast as a pre-determined rate under specific workloads [19] [20].

B. Bad and good test case design comparison

We used two artefacts in our analysis: the test bot source code and the test cases being executed. From the aforementioned artefacts, we were able to discover two different test bots which depicted the differences between good and bad test case design. However, since the two bots tested out different system functionalities, we decided to create a new version of the test bot in coordination with the colleagues which was following the bad test case design. This newly developed bot contained test cases which tested out the same functionality as the original one. The second version (bad design) was benchmarked locally on one development machine instead of on the actual system's environment. To ensure that we get valid results which are not dependant on the response time from the system endpoints, each functionality was timed from the actual system's environment and later the duration's were

TABLE III
DIFFERENCES BETWEEN TEST BOT VERSION ONE AND TEST BOT VERSION TWO

Test bot	Framework	Programming method
Version 1	JUnit	Synchronous
Version 2	In-house developed test runner	Asynchronous

added as timeouts in the test cases in the bad test bot. We used a library called WireMock to mock the responses from the system, the test cases developed were validated to make sure the design was bad by iterating with practitioners working in the projects. Even though the different versions tested the identical functionalities of the system, there were factors which made it clear which one of those test design approaches was more beneficial. The underlying factors were: i) execution time, ii) false positive results and iii) complexity of the test cases. To better understand how these factors influenced the whole system testing we will describe how they affect the system test results and other important variables.

In terms of test execution time, the shorter the time the better, thus results will be displayed faster while the cost will be lower, as these automated test bots are usually hosted on server-less applications which have their own pay per use model, which takes the milliseconds needed to run the application as a baseline for the cost.

In one of the test bot versions more false positives or flaky test occurred due to problems with the test dependencies, these issue were generated because of the difference in the test design, which in that case was using different programming methods and frameworks to execute the test cases. Cyclomatic complexity is a software metric which depicts the complexity of a program, it is a quantitative measure of the number of linearly independent paths through a program's source code. There were also differences when it comes to the cyclomatic complexity of the test cases in combination with the lines of code. In table III the specific test bot versions are compared and the underlying differences which affect the previously mentioned factors are depicted and described in the following subsections.

C. Synchronous vs asynchronous programming

Test bot version 1 uses synchronous programming methods to call the REST endpoints while version 2 uses asynchronous methods for the same purpose. In programming, synchronous operations block instructions until the task is completed, while asynchronous operations can execute without blocking other operations. Asynchronous operations are generally completed by executing an event or by calling a provided callback function. In the context of system testing when using synchronous methods in each test case the first test will block the second one from executing and that linear dependency will continue until the last test, this will affect the execution time of the test bot since it will gradually increase as it goes from test to test. On the other hand when using asynchronous request the test case number one can invoke the endpoint that it needs to test out, but also in the same time the other test will be invoked within their own separate thread, which will utilize the power of modern day central processing units (CPUs) that contain multiple cores. A thread can be described as the smallest sequence of programmed instructions that can be managed independently by the operating system's scheduler. This is especially important when the test suite contains test that take a long time to execute. Asynchronous programming allows the test bot to initiate multiple tests in parallel and it handles the result later when the response payload has been delivered by the micro-service. This type of programming increases the efficiency of the test bot by a big margin when it comes to the time that it needs to execute the whole test suite. There are multiple frameworks that exist for asynchronous programming in this project the development team has been using the built in Java library CompletableFuture, which contains a large number of functions that allow them to tailor the test suite to be as efficient as possible.

To get a better understanding of the CPU power utilization when using asynchronous programming, we will describe an ordinary scenario of using a two core processor. For example if the test suite contains four test and it uses asynchronous methods to handle the request towards the system, the first two test when initiated will live within their own unique thread and those threads can be specified to run in parallel on the two CPU cores. With this approach if hypothetically speaking each test takes two seconds to execute the total time of execution for the two test will be two seconds, unlike the synchronous approach where the total time of execution would be four seconds, because the test would be executed one after another using the resources of only one thread [21].

The second version of the test bot which used the asynchronous request handling method was found to be more efficient based on the previous explanation, to prove this we compared the test execution times between the two separate test suite versions. It is worthy to mention that the test suite tested the identical functionalities of the system with the same order, the only differences was the programming method used. To perform this benchmark we used the start time or the time when the testing has been initiated and the time-stamp from the test report which reflected the time when the testing has been finished. There was a large difference between the two test suites where it was observed that version two (*"good test bot design"*) took 12 seconds to finish executing as compared to version one (*"bad test bot design"*) which took 20 seconds. Therefore, according to the results it can be stated that version two performed (40%) faster than version one, as was expected.

D. JUnit vs in-house built test runner

The second big difference between the two test bot versions is the usage of different testing frameworks. In version one the test suite is executed using JUnit and in version two the test suite is being executed by an in house built test runner. JUnit can be used as a test runner for any kind of test: system and integration tests; tests which are interacting with a deployed application. However the reason for using a homemade test runner is because it gives more flexibility

and it can be designed as wanted, instead of following the principles and methods that another framework provides. One huge benefit of using a built in test runner is the rules that can be set by asynchronous programming. For example if the test suite contains test which have dependencies to other test, but also has test that can be executed without any dependencies, with the asynchronous framework, rules can be enforced that would immensely influence the execution time, as explained in the previous section.

On the other hand, when using JUnit to achieve these modifications it can be quite hard or even impossible, and that is mostly because the test in JUnit are not aware of the execution process of other threads which would be the case when using asynchronous request to invoke an endpoint in the same test case. To put it simply, if we take a case where we would have test B which depends on test A to finish, and also take into consideration that we would use specific JUnit annotations where we state the order of the test executed, in that scenario test case A would finish executing before the request to the endpoint has been completed, and in that time period when test B is executed it will give a failed result, which can be classified as a false positive, because the test failed from an incomplete test dependency.

Two mitigation strategies can be used when combining JUnit with asynchronous programming to avoid the aforementioned issues, one is to actually block the asynchronous call which is executed in test A, and this defies the purpose of using the specific programming method since it will behave like a synchronous call. The second fix that can be applied is to use a timeout annotation which is provided by JUnit to make test case B idle for the specified amount, this strategy will also affect the execution time negatively and in the worst case even more than when using synchronous calls. Another comparison can be made in the design of the test cases from the two versions is that to fulfill the dependencies, test in version one use synchronous calls to the endpoints one after another, this is to ensure that problems like the one explained previously with false positives do not occur. Even though this approach works, it blocks all the other test cases from executing which do not contain any dependencies and in this case the test case size and complexity increases marginally, which does not follow the approach of modular and understandable test cases, as stated by the participants. To discover the difference between the complexity of the two unique test, we performed an analysis on the software artefact when it comes to the lines of code in a combination with the cyclomatic complexity.

The test cases in version one of the test bot contained on average 71 lines of code per test case, while version two contained 51 lines of code, meaning that the average lines of code was (28%) higher in the badly designed test bot. In terms of the average cyclomatic complexity per test case, version one was measured to be (33%) higher than version two. The average cyclomatic complexity per test case was found to be 9 and 6 in version one and two respectively.

VI. Discussion

A. Guidelines

Using the data collected and outlined in the previous section Test bot challenges, we created a table IV that outlines the guidelines that we believe will help mitigate these challenges. The table consists of 7 guidelines for creating good system case design, each labeled with an ID, the reason behind the guidelines along with the suggestion on how to implement it. The purpose of the guideline is to make it easier for developers designing test cases to make their test bot more efficient.

B. Answers to research questions

RQ1: What are some of the challenges and practices when designing and executing system tests on software bots?

During the interviews, participants discussed the issues that can appear with the design and execution of the test cases, from this data together with a cross evaluation with existing literature we were able to answer our first research question. Some challenges that can be associated with the system test cases are: i) designing context aware test, ii) changes in the state of the system, iii) system rollbacks and iv) false positive or flaky results. For all of the aforementioned points we have a thorough description within the test bot challenges section.

RQ2: To what extent does the test design affect the effectiveness of a test bot?

Our second research question was related to the effects of the test case design on the effectiveness of the test bot, we can state that the design of the test cases severely influences the results that are produced from the test bot, this claim can be supported by the answers from our two sub research questions. The main factors that changes when having different design approaches of the test cases are: i) test bot execution time, ii) complexity of the test cases, and iii) the number of false positive test results. How these factors influence the effectiveness of the test bot has been described in the previous section.

Our first sub research question focuses on the differences between good and bad test case design, the answer to this question was derived from a comparison between the two test bot versions that tested the identical functionalities of the system. We have discovered that using asynchronous programming methods lead to a better test case design by decreasing the execution time and solving some complex test dependencies scenarios. When using the aforementioned programming method there can be issues when trying to incorporate it within existing testing frameworks, so a prominent adoption that we have discovered in this scenario is to create a simple test runner class which would handle all the test cases, together with the creation of the test reports.

The second sub research question was related to the guidelines on how to create good test design, and this was answered by analyzing the test suite version which followed the good test design which in turn gave us better benchmarking results when it came to execution time and test case complexity

TABLE IV
GUIDELINES FOR CREATING GOOD SYSTEM TEST CASE DESIGN

ID	Description	Reason behind guideline	Suggestion on how to implement it
G1	Ensure completeness of system specifications and or documentation.	Testing a functionality of a system might result in positive results, in accordance to the function's purpose, however it might not take into consideration the system specification, and thus developers might miss an important testing aspect.	Include trace links between test cases and user documentation (e.g., include a Jira item ID into the test case).
G2	Use asynchronous programming methods to invoke system endpoints.	According to the data analyzed, asynchronous programming methods allows the tests to continue testing independent functions simultaneously and can thus reduce execution time.	With the Java framework CompletableFuture test can be executed with the method. `supplyAsync()`, in this case the framework will run the task asynchronously and return the result from the test without blocking the execution of other test.
G3	Cover test dependencies by chaining dependant test via the specific callback asynchronous functions.	For dependant tests, tests which require the completion status of previous tests, this would allow the system to wait before executing the next tests, while for independent tests, these can be unchained thus allowing all the non dependant test to run within their order.	With the Java framework CompletableFuture test A can be executed with the method `supplyAsync()`, and test B needs to be chained to the first future using the function `thenComposeAsync()`, this way test B will execute once A has finished without blocking other test executions in the test suite.
G4	Create test that are small, modular and readable.	By creating smaller and more readable tests, developers can ensure that future alterations to the tests are easier to implement, as well as less time will be needed to analyze the existing setup and develop the new test.	Create test cases that test a specific functionality of the system rather than multiple flows.
G5	Start clean-up process after the execution of tests.	In order to avoid adding corrupt or unnecessary test data to real environments, developers should include a clean-up process after the execution of tests within the test bot.	Use existing (or implement) functionality to remove data which was stored in the system by the test suite (e.g., after testing the user creation functionality, use the delete user service to remove the test data).
G6	Make use of proper logging techniques which differentiate the test data from real data.	In order to make it easier to distinguish the *test activity logs* from the actual *user activity logs*.	Use different prefix for test data attributes, thus making it easier for developers to distinguish test data from real data in real environments (e.g., username starts with "TEST").
G7	Implement both positive and negative flow testing techniques.	According to Laventhal et al. [9], software testing theory suggests that tests should test inside and outside specification (expected versus unexpected values) in order to test thoroughly.	Use both invalid and valid test data as input.

(lines of code and cyclomatic complexity). We have outlined the guidelines (see table IV) which we believe should help developers create a good test case design, which were as follows;

1) Ensure completeness of system specifications and or documentation.
2) Use asynchronous programming methods to invoke system endpoints.
3) Cover test dependencies by chaining dependant test via the specific callback asynchronous functions.
4) Create test that are small, modular and readable.
5) Start clean-up process after the execution of tests.
6) Make use of proper logging techniques which differentiate the test data from real data.
7) Implement both positive and negative flow testing techniques.

C. Contrasting results with other study

In the findings from the Canadian study by Garousi and Zhi [22], they point out a couple of factors which can be contrasted with our research, however worthy to note is that since the survey questions were mostly different, we can only compare a small portion of the findings. According to the research in the Canadian study, maintainability of large or complex test suites was mentioned to be a challenge by the respondents. This correlates with our findings where one of the interviewees mentioned that tests are too complicated due to their need to be refactored, thus making it harder to maintain the test.

D. Threats to validity

According to Wohlin et al. [23] there are four main types of validity threats: conclusion, internal, construct and external. Of these four validity threats, we outlined three of them that we believed were mostly relevant to our study.

Internal validity: As the interview questions were used as a base line, any deviations resulting from rephrased questions could have resulted in different understandings by the interviewees and therefore it was important to make sure that the questions placed were consistent across all the other interviews. To mitigate this internal validity threat, we had to therefore state the original question along with the rephrased version during the interview for all participants. Another internal validity threat that was identified was the "poorly designed" test bot version which was developed by the authors. It could be argued that this could lead to research bias, which is a process where the researchers influence the results in order to portray a certain outcome. However, to mitigate this threat the previously mentioned test bot version was developed

in coordination with the practitioners that were part of the development team. The test bot which was developed by us was following a test case design from a bot used in a different inactive project. This design was deemed as unsuitable from the practitioners due to the many issues it created during the testing process in the past. There are two factors why this test bot was developed by the researchers, the first and more important one is that due to the lack of time we were not able to interview developers from other projects within the company. If we had more time we could have been able to discover a version of a test bot which has been evolving and where many design issues have been fixed over a specific time period in order to make the test bot more efficient. The second reason is that the chosen project was relatively new when compared to other more mature ones in the organization, hence the fact that a lack of evolving test bot versions were found. The team brought their experience from previous projects to fine-tune the design of the test bot by avoiding past design issues which led to the lack of test bot version iterations that could have been used in this study for the comparison. This project was chosen by convenience, mainly because it was the only project where the developers accepted any interviews that actually coaligned with our schedule. This type of selection could lead to inclusive bias, where the participants are selected for convenience, however we have mitigated this threat by stating in our report that the results can not be extrapolated to fit the entire software engineering industry, moreover it is worth mentioning that more research needs to be performed on this topic outside of the automotive IT industry.

Construct validity: One threat that can be stated within the construct validity is whether the questions from the interview instrument were good enough to capture the correct information from the practitioners. To diminish this threat we initially used two methods, face validity and a small pilot study. Face validity can be described as a lightweight review by an investigator outside the study to give initial feedback [24], which in our case, the reviewers were our thesis supervisors, and the feedback that we got helped to us to modify and adapt some of the questions. The pilot study was performed with the help of two colleagues from the university, who had some previous experience with test bots and test case design. Some of the questions used for this survey were based on Garousi and Zhi study which analyzed software testing practices in Canada [22].

External validity: As this study was performed within an organization, the data collected and analyzed was relative to the way of working of the organization and as such it was therefore hard to generalize in order to make the results viable for other organizations. Furthermore, as the researchers were currently working at the organization, this could have caused researcher bias, which was mitigated by having close coordination with our supervisors who were external to the organization and thus not employed or affected by the bias.

VII. Conclusion

In this thesis we have outlined the current challenges in designing test cases for system tests executed by a test bot and the issues that can occur when using these tests on a system which is simultaneously used by customers. This paper also provides information on how the test design affects the effectiveness of the test bot and also depicts differences between a good and bad test case design. Finally, we propose guidelines on how to create test cases for system tests that are executed on a test bot. The guidelines proposed are based on industrial data gathered from different sources: i) interview with industry practitioners, ii) analysis of software artefacts (test bot code and corresponding test cases), iii) analysis of a single test execution and system documentation.

The case study company works in the field of automotive cloud and develops services for car manufactures around the world. Four interviews have been performed within the organization with developers of different expertise.

Most of the guidelines proposed in this study can only be used when developing system tests that are executed on an actual system environment (i.e., a system being accessed by the end-users while test is ongoing). For instance, system testing can also be executed on a system test machine and in a specially configured environment that simulates the end user environment as realistically as possible [19]. In this scenario some of the guidelines can not be used since they are limited to the scope of system testing where the actual system that is used by customers is tested. Due to the time constraints we were not able to analyze different test bots which were a part of projects that are were developed in different programs for other customers. Since customer specification vary from project to project, we could have generated more findings on the challenges, mitigation strategies and guidelines associated with system testing using bots.

Future work can be done on the study of test design by exploring fields outside of the automotive branch. Possible future work could include the study of the correlation between IT fields and practices of testing to help create a more generalized guideline which can be applied across different IT industries.

References

[1] W.-T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE, 2001, pp. 166–171.

[2] A. Elssamadisy and J. Whitmore, "Functional testing," *Proceedings of the 2006 conference on Pattern languages of programs - PLoP 06*, 2006.

[3] I. Hooda and R. S. Chhillar, "Software test process, testing types and techniques," *International Journal of Computer Applications*, vol. 111, no. 13, p. 1014, 2015.

[4] L. Erlenhov, F. G. de Oliveira Neto, R. Scandariato, and P. Leitner, "Current and future bots in software development," in *First Workshop on Bots in Software Engineering, (BotSE ICSE)*, 2019.

[5] C. Lebeuf, M.-A. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, p. 1823, 2018.

[6] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009.

[7] S. Matalonga, F. Rodrigues, and G. H. Travassos, "Matching context aware software testing design techniques to ISO/IEC/IEEE 29119," in *International Conference on Software Process Improvement and Capability Determination*. Springer, 2015, pp. 33–44.
[8] P. Oladimeji, M. Roggenbach, and H. Schlingloff, *Levels of testing*. Advance Topics in Computer Science, 2007.
[9] L. M. Leventhal, B. E. Teasley, and D. S. Rohlman, "Analyses of factors related to positive test bias in software testing," *International Journal of Human-Computer Studies*, vol. 41, no. 5, pp. 717–749, 1994.
[10] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.
[11] U. Schultze and M. Avital, "Designing interviews to generate rich data for information systems research," *Information and organization*, vol. 21, no. 1, pp. 1–16, 2011.
[12] O. Doody and M. Noonan, "Preparing and conducting interviews to collect data," *Nurse Researcher*, vol. 20, no. 5, p. 2832, 2013.
[13] M. K. Smith, "Chris argyris: theories of action, double-loop learning and organizational learning," Nov 2013.
[14] M. Maguire and B. Delahunt, "Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars." *AISHE-J: The All Ireland Journal of Teaching and Learning in Higher Education*, vol. 9, no. 3, 2017.
[15] D. R. Thomas, "A general inductive approach for analyzing qualitative evaluation data," *American journal of evaluation*, vol. 27, no. 2, pp. 237–246, 2006.
[16] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, p. 77101, 2006.
[17] D. Firesmith, "Common testing problems: Pitfalls to prevent and mitigate," in *A1AA Case Conference*, 2013.
[18] V. Nguyen, V. Pham, and V. Lam, "Test case point analysis: An approach to estimating software testing size."
[19] P. Oladimeji, "Levels of testing," 2007.
[20] L. Briand and Y. Labiche, "A uml-based approach to system testing," *Software and Systems Modeling*, vol. 1, no. 1, pp. 10–42, 2002.
[21] J. Manson, W. Pugh, and S. V. Adve, *The Java memory model*. ACM, 2005, vol. 40, no. 1.
[22] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, p. 13541376, 2013.
[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
[24] J. Linker, S. Sulaman, M. Host, and R. de Mello, "Guidelines for conducting surveys in software engineering," 05 2015.

APPENDIX

A. Interview Questions

1) What is your primary work position (job title)?
2) What are your main responsibilities as a (#jobtitle)?
3) How many years of work experience do you have in IT and in software development industries?
4) How many years of work experience do you have in software testing?
5) Which programming languages are used when writing your test cases?
6) What kind of testing framework are you currently using?
7) In your current or most recent project, what are the steps you follow when designing test cases?
8) What information do you use to design the tests?
9) Where do you get this information?
10) How do you write the assertions of the tests (ie, determining that a test passes/fails)?
11) Do you currently use any test bots ? (If yes on previous question) What kind of testing do you perform (ie, unit, integration, load, system testing) when using the test bots ?
12) How would you describe the role of the test bot (the job it has to fulfill, purpose)?
13) What do you have to take into consideration when developing test cases that are executed by test bots?
14) Do you have any standard practices when designing test cases?
15) What is considered a good test case design?
16) What is considered a bad test case design ?
17) What impediments have you or your team experienced when designing test cases for the test bot?
18) How do you currently record the test results? What is the current practice used to gather test bot results and display it for developers?
19) Do you think that the test results are meaningful in providing feedback? (Yes/No) (In both cases) Why?
20) Would you be open to trying out different techniques for relaying test bot results to developers? If Yes: What types of techniques, or information, you would like to present to developers?

B. Re-Used Interview Questions

The following interview questions have been re-used [22]:

1) What is (are) your current position(s)?
2) How many years of work experience do you have in IT and software development industries?
3) How many years of work experience do you have in software testing, in specific?
4) Which programming languages do you use in your company?
5) Which test automated tools/ frameworks do you use in your company?
6) Please provide a list of top 3 testing challenges that you have been seeing in your projects and you would like the software testing research community to work on. (For example, you might say: the current record/ playback GUI testing techniques/ tools are not very stable and need to be improved).