Bachelor Degree Project

Large Language Models for Unit Test Generation in React Native TypeScript Components

Author: Erik Borgström
Author: Robin Bergvall
Supervisor: Prof. Dr. Welf Löwe
External supervisor: Tibo Bruneel
Examiner: Dr. Nadeem Abbas
Semester: VT 2024
Subject: Computer Science

Abstract

Advancements within Large Language Models(LLMs) have opened a world of opportunities within the software development domain. This thesis, through an controlled experiment, aims to investigate how LLMs can be utilized within software testing, more specifically unit testing. The controlled experiment was performed using a Python script interfacing with the gpt-3.5-turbo model, to automatically generate unit tests for React Native components written in TypeScript. The pipeline described, performs the calls to the OpenAI Application Programming Interface(API) iterative. To evaluate and retrieve the metric code coverage, the unit tests were executed with Jest. Additionally, manual execution of failing tests, both compilable and non-compilable tests were executed and the different kind of errors with their frequency were documented.

The experiment shows that LLMs can be used to generate comprehensive and accurate unit tests, with high potential of future improvements. While the amount of generated tests that compiled were low, their nature was often good, failing because of easy correctable syntax errors, faulty imports or missing dependencies. The errors found, were at large part due to project configurations while others would probably be less frequent through more extensive prompt-engineering or by the use of an newer model. The experiment also shows that the temperature affected the outcome and that the type of errors were different between compiling and non-compiling tests. A lower temperature parameter to the OpenAI API generally achieved better results, whilst a higher temperature showed greater coverage at compiled failing tests. This thesis also shows that future opportunities and improvements are widely available. Through better project optimization, newer models and better prompting, a better result is to be expected. The script could with further development be turned into a working product, making software testing faster and more efficient, saving both time and money while simultaneously improving the test case quality.

Keywords: Unit testing, Large Language Models, AI, ChatGPT, gpt-3.5-turbo, Software testing, React Native, TypeScript.

Preface

We would like to thank the many people involved making our thesis possible. First we want to give our most humble thanks to our thesis supervisors Prof. Dr. Welf Löwe and Tibo Bruneel for their extensive work, providing us with a lot of good concrete feedback, ideas and support in the different directions presented during the thesis work. Both Tibo Bruneel and Prof. Dr. Welf Löwe have inspired us to reach higher and without their critical feedback and tons of suggestions this thesis would not be what it is today.

We would also like to thank Björn Lundsten and Anthony Mulot for making this thesis possible, providing us with the InfoSynk codebase and resources, making the experiment possible. Lastly we want to thank the staff at Softwerk AB for welcoming us, making it an exciting place to spend the time whilst writing this thesis.

And lastly to you who are reading this, thank you for taking your time reading this thesis, and hopefully it gives you as much knowledge reading it as we got from writing it.

Erik Borgström & Robin Bergvall
18-May-2024

Contents

1 Introduction

This is a 15 ECTS Bachelor thesis in Computer science for Linnaeus University. It investigates the field of Large Language Models (LLMs), specifically OpenAI's model gpt-3.5-turbo and its application within software development. LLMs trained on massive amounts of data, have the ability to create human-like text and perform complex tasks. Within software development, much research is dedicated to exploring the different application areas where the use of LLMs can be useful. One such area is software testing, and one way to test software is through unit tests.

Unit testing is a core practice within software testing and is performed by testing individual units or components in isolation with the goal of validating their correctness and functionality [1]. In this study, we aim to investigate the potential of LLMs in generating test cases for React Native components written in TypeScript.

1.1 Background

Software development includes a broad range of activities where computer science techniques are used to create, maintain, and enhance software systems. Within this domain, software testing plays an important role, ensuring the reliability, functionality, and security of software applications. Unit testing is a well-used technique within software testing and involves testing individual units or components in isolation with the goal of validating their correctness and functionality [1].

Our research focuses on the utilization of Large Language Models (LLMs) within software testing, more specifically towards LLMs ability in generating unit tests for React Native components written in TypeScript. Previous research performed in the field of software testing and the use of LLMs for the generation of unit testing is well-documented for languages such as Java, Python, and Javascript [2, 3, 4]. There is however a notable research gap with its application on code written in the React Native framework using Typescript.

1.2 Related work

Unit testing is an area with a lot of research being made, where LLMs ability to help have been focused on since the rise of better AI models. According to Wang et al. [5], the amount of publications in the research area has increased from one article in 2020, to 82 articles as of last year. [5]. According to Daka et al. [6], previous research have shown that code testing is a labor-intensive task that more often than not is skipped by the developer if time is crucial. LLMs have been used to generate test cases for several languages such as Javascript but not TypeScript together with the React Native framework. The work done [2] shows that LLMs test generation outperforms previous state-of-the-art test generation methods such as Nessie [7] and that the use of more advanced LLM should only enhance this. This research is close to our field and serves as a good base and as it is recently published it is highly relevant [2]. Another article authored by Wang et al [5] shows that LLMs have been applied to a wide range of testing tasks and have been shown useful. It does come with challenges though that needs further research. The study relates to our research as we will explore LLMs ability to test code and as the article is relatively new the data is not outdated [5].

1.3 Problem formulation

The framework React Native with the language Typescript is growing fast in popularity and in need of faster and easier testing. Typescript and React Native testing capabilities using LLMs haven't been researched and are in need of a more efficient way to generate test cases. By addressing the following research questions the knowledge gap can therefore be narrowed:

1. *Can LLMs be used to generate comprehensive and accurate unit tests for React Native components written in TypeScript?*
2. *What impact do different temperatures have on the generated unit tests?*
3. *What type of problems are detected when generating unit tests using LLMs?*

1.4 Motivation

Writing unit tests is a tedious, time consuming and costly process in the software development workflow. Research from 2019 provides that on average, 23% of a organization's annual IT spending's is dedicated towards quality assurance and testing [8]. There is also multiple challenges specific to automatic GUI testing, making the process harder. [9]

This research aims to improve front-end testing by providing an effective way of generating tests. By providing more knowledge on how to utilize LLMs we hope to increase its usage and thereby also increase the amount of front-end testing performed, streamlining the process.

Through the first question we aim to address the core functionality of LLMs in generating unit tests, it helps us understand LLMs capabilities and limitations better. The second questions explores what effect the parameter temperature has. By exploring the impact of different temperatures we can fine-tune the LLM, providing valuable intel for further research. The last question can help developers understand the limitations and potential faults when using LLMs to generate unit tests.

1.5 Results

As described in chapter 5. By using our script on the provided private project InfoSynk from Softwerk AB, we show that LLMs can generate comprehensive and accurate unit tests for React Native components written in TypeScript.

By generating over 1000 different tests, the results shows us that comprehensive and accurate tests can be generated by the help of LLMs. All the test that was generated was well written and easy to understand. This shows that LLMs can clearly help with testing future software in React Native Typescript.

By generating tests with the help of several different temperatures we show the difference of what the temperature can do. We provide a clear understanding of what the difference is and how it affects the generated tests.

We also evaluate the tests by running them manually, non-compiled as well as compiled to get a clear understanding of what problems have occurred. We provide graphs over the problems and a discussion on how these may be avoided.

1.6 Scope/Limitation

The scope of this thesis is limited to React Native components written in TypeScript. A more general approach testing multiple front-end configurations could have been performed, but is not within the scope for this thesis. The thesis is also limited to gpt-3.5-turbo, but future work could benefit from testing of other models as well. The thesis work is also limited to the private system provided by Softwerk AB, called InfoSynk. This limits the thesis to only one source for the testable data, and also makes the inclusion of the generated tests impossible. The code retrieved from InfoSynk is deployed, allowing us to assume that it is correct. Therefore, we only test on code that is presumed to be correct. Including incorrect code that's knowingly wrong could be beneficial to the experiment, allowing for a comparison of LLM performance between correct and incorrect code, but outside the scope for this thesis.

1.7 Target group

This thesis work is targeted towards developers utilizing the React Native framework as well as data scientists researching large language models. Front-end developers that use other frameworks may also benefit from this thesis.

1.8 Outline

Our research report is structured into the following chapters. In chapter 2 we describe the theoretical area and dive into the theoretical background needed to understand this report. Topics such as React Native, Unit testing and Large Language Models will be explained. Chapter 3 explains and discusses our method used to answer our research questions, what tools we used, and how we will utilize them. Chapter 4 describes the design and implementation, here we will discuss our implementation of our script used to generate automatic tests. In chapter 5 we show our results, the data we gathered, and how well the LLM performed on different metrics. In chapter 6 we analyze our results provided in chapter 5. In chapter 7 we discuss the results and answer whether our research questions were answered by this research. Lastly in chapter 8 we make a conclusion of our work and provide an idea of future research work.

2 Theoretical Background

This chapter discusses the essential theoretical background needed for the research presented in this paper.

2.1 Large Language Models

Large Language Models (LLMs) are advanced computational models that are designed to do complex tasks such as generating and understanding human language. The application of LLMs is large such as language generation, translation, text classification and much more. The models are trained by the use of a large amount of data to learn patterns, relationships and structures [10].

Large Language Model is a computational model with the ability of general-purpose language generation and other tasks such as classification. LLM learn its abilities from statistical relationships found in text documents, repeatedly predicting the next token, based on the statistical relationships [10].

2.1.1 Transformers

LLMs leverage deep learning architectures, and one of the most prominent architectures is known as the Transformer. The Transformer is a architecture developed by Google, originally proposed in "Attention Is All You Need" [11]. It utilizes self-attention to learn input sequences efficiently.

In the Transformer architecture, input text is tokenized into discrete units known as tokens. Each token is then converted into a vector obtained from learned embeddings, transforming the textual data into vector format, suitable for deep learning models. The Transformer uses its self-attention, to dynamically weigh the importance of different tokens within the input sequence, capturing dependencies and relationships [11, 12].

Within a transformer, one of the crucial components is the decoder network. This network consists of multiple layers. In these layers, matrix multiplication and softmax operations are performed. These operations compute probabilities for each word's chance of becoming the next word in the sequence. The transformer analyzes the relationships between different words through these techniques, enabling it to predict the next word accurately [13, 12].

It is important to note that these operations are also extensively used in the encoder network, which, along with the decoder, forms the complete structure of a transformer. The encoder network plays a significant role in processing and understanding the input sequence, often involving more matrix multiplication and softmax operations compared to the decoder [13, 12].

The Transformer architecture is applied across various domains, including natural language processing(NLP), sequence learning, audio processing, and multi-modal learning. It has served as the basis for developing pre-trained systems such as Generative Pre-trained Transformers (GPTs) and other models [14].

2.1.2 Softmax

The softmax function is used to turn a vector of K real values into a vector of K real positive values that sum to 1. It's responsible for presenting the output vector as a vector with probabilities. The vector with probabilities is then used for the LLM to choose its next token to present [14]. The formula for the function is defined as the following:

$$\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax
$\overrightarrow{z}$ = input vector
e^{z_i} = standard exponential function for input vector
K = number of classes in the multi-class classifier
e^{z_j} = standard exponential function for output vector

A hyperparameter T, temperature could also be applied to the softmax formula to be able to fine tune it. The formula transforms into the following:

$$\sigma(\overrightarrow{z})_i = \frac{e^{\frac{z_i}{T}}}{\sum_{j=1}^{K} e^{\frac{z_j}{T}}}$$

T = temperature

The standard value for T is 1, but can be altered to change how the probabilities is distributed within the output vector. A higher temperature value as explained in Section 3.4, results in a more even distribution between the probabilities for each token, resulting in a more "random" result. In comparison, a lower value decreases the diversity and makes the choice of the next token more deterministic, resulting in more predictable and conservative outputs.

2.2 React Native

React Native is a framework[1] used to build native cross-platform applications. Using React Native enables the developer to use a single code base for web, Android and iOS but still leverage the performance gained from using native components, as React Native uses it under the hood.

2.3 TypeScript

TypeScript is much like JavaScript, a powerful language that according to Github [15] is the fourth most popular language used, only conquered by JavaScript, Python and Shell. According to David Choi [16] the reason for TypeScripts growth is due to the limitations JavaScript reveals when developing large applications. Developers quickly realised that JavaScript was limited in OOP development, and that dynamic typing that is used in JavaScript is not suited for bigger applications. These limitations resulted in the creation of TypeScript.

TypeScript is not only a language as it is also a compiler. It uses transpilation, compiling the TypeScript files into JavaScript when building a solution. By using TypeScript developers can easier develop structured code, promoting the use of OOP and static typing. By using static typing the developers can define types for variables, parameters and return parameters which tend to help in catching errors during the development. [16]. OOP becomes easier to follow, JavaScript does not support interfaces and abstract classes meaning that inheritance is much more powerful using TypeScript. TypeScript also adds the access modifiers private, public and protected.

[1] https://www.geeksforgeeks.org/what-is-a-framework/

2.4 Unit Testing

A valuable practice for developers testing software is unit testing. A unit test is according to Vladimir Khorikov [17] defined as an automated test that verifies a small piece of code, and does this quickly and in an isolated manner. The first two attributes are non disputable but different opinions about the third attribute has lead to two different styles, classical and London. The difference of the schools is portrayed in the way they handle the isolation issue. The London approach replaces dependencies by dummies, also known as mocks, while the classical way does not [17].

The main goal of unit testing is to provide projects a good way of growing. Through the tests new functionality and refactoring can be performed without introducing regression. The book specifies two main benefits of unit testing, its ability to warn early when existing functionality is broken and providing confidence that code changes wont lead to regression [17]. Per Runeson has also researched unit testing and describes the main goal of unit testing as a way of ensuring a systems functionality [18]. It also makes an important point that the assurance is towards the developers expectations of how the system should behave and function, not other stakeholders [18].

2.4.1 Test case structure

There are multiple ways to structure a unit test, but its usually represented after the AAA pattern Arrange, Act and Assert. The Arrange phase is used to set up the necessary conditions and inputs for the test, the Act phase involves executing the behavior that you want to test and the Assert phase checks that the expected outcomes have occurred. There could be multiple of these sections within a test but would then transform the test into a integration test and should therefor be avoided. Multiple assert statements within a unit test is often discouraged, but according to Vladimir Khorikov these statements are wrong [17]. A unit in a unit test is defined as a unit of behavior and not a unit of code. Multiple behaviors could therefor be tested and observed within a single unit test, arguing for the ability to have multiple assert statements within a single test [17].

2.4.2 Testing environment

There are many different testing frameworks available on the market today. Some well used tools when creating unit tests are for example JUnit[2], Jest[3] and Jasmine[4]. These frameworks are designed to allow for easy creation and execution of unit tests, allowing developers to ensure the correctness of code in isolation. Jest, which is a framework built on top of Jasmine also allows for integration and snapshot tests.

Selenium[5] on the other hand is not a framework but rather a set of frameworks bundled into a suite. Its primary use is to create and run tests at the UI layer of a web application, also known as UI testing. It supports languages such as C, Java and JavaScript, and it can be used together with for example Jest to automate functional tests on websites and web applications. Each of the testing frameworks provides the developer with rules and tools. Through the use of the frameworks developers can efficiently create, manage and execute test on their code, assuring that its correct and of high quality.

[2]https://junit.org/junit5/
[3]https://jestjs.io/
[4]https://jasmine.github.io/
[5]https://www.selenium.dev/

3 Method

This research project has been created to answer the different research questions and knowledge gap identified in this paper. This chapter will provide the description of the project together with the methods used to answer the research questions and to cover the knowledge gap.

3.1 Research Project

This research project explores the use of LLMs in generating unit tests for React Native components written in TypeScript. A script will be developed to automatically generate unit tests for the React Native components. It will also generate statistical files containing data to be analysed.

To answer the research questions we will perform a controlled experiment [19]. In the experiments we will have independent and dependent variables. The hyperparameter temperature, used when prompting gpt-3.5-turbo is an independent variable. It's the only variable we intentionally vary during the experiment. The metrics used to assess the quality of the unit tests, explained in Section 3.3.4, are examples of dependent variables. These are what we observe changing in response to the manipulation of the independent variables.

The research will be performed in collaboration with Softwerk AB which will provide testable React Native components from one of their existing private codebases. Further development is performed on the product InfoSynk to generate more React Native components and thereby testable data.

3.2 Research methods

To answer the research questions, we did a quantitative study using experiment method as described in *Experimentation in Software Engineering* [19]. In the experiment, by analyzing the statistics and different temperatures in detail, we got a good understanding on how well an LLM can generate unit tests, how different temperatures affects the outcome and the quality of the unit tests generated.

The experiment was performed through the use of the model gpt-3.5-turbo. It generated test cases for React Native TypeScript components. From the experiment we was able to retrieve metrics described in Section 3.3. This process was repeated ten times, improving validity as well as reliability. From the data we were able to evaluate the quality of the test cases, the models ability to generate test cases and how the hyperparameter temperature described in Section 2.4 affects the result, answering our first two research questions. The second part of the experiment was then performed by manually executing the failed tests generated in the first part. Both non-compilable and compilable tests was included, and after execution the errors was documented. From the data concluded in the second part of the experiment we were able to detect what type of problems occur when generating and running unit tests, and how errors in non-compilable and compilable unit tests differentiate, thereby answering our third research question.

3.3 Evaluation metrics

In this chapter we will discuss the different metrics that will be used in this study for the evaluation of gpt-3.5-turbo.

3.3.1 Passing rate

Passing rate is a metric that tells us the percentage of passing tests. For each of the 49 components we try ten times to generate a passing and compilable test and then set true or false. After all tests have been generated for all components we measure how many components have a passing test. It's a crucial metric in determining the LLMs generative ability, and by analyzing it we can also determine if different temperatures described in Section 2.3.5 impact this.

3.3.2 Code coverage

Code coverage is a metric used to measure how extensive a unit test tests the code [20], used by many companies as a control before allowing the developers to deploy or upload their code. The study will measure branch coverage, line coverage, statements coverage and function coverage, all of which is provided by the test library Jest. The code coverage will be generated after each test and will only be measured for the component under test. Line coverage measures the percentage of executable code lines that the test executes. Branch coverage measures the percentage of tested decision points in the code. A decision point is for example a "if statement" or a "loop". It checks both for negative and positive condition of a decision point. Statement coverage is much like line coverage but takes multiple statements in a single line into consideration as well. Function coverage is the percentage of functions within a components, that the unit test tests [20].

3.3.3 Change frequency

Change frequency measures how many times it will take for the LLM to generate a passing test. The tests will be generated by the LLM and if it's unable to compile, the failed test, error messages and the original message will be sent back. After 5 tries the generation will end if none of the tests generated has managed to pass or compile.

3.3.4 Test-case Quality

Test case quality is measured through the previous named metric code coverage. It's hard to evaluate how well a unit test tests a component quantitatively. The metric mutation score has been used in previous research [21] relatable to our work, but unavailable to this thesis. Mutation score measures the amount of bugs a unit test discovers, but to measure it you need access to faulty data with documented faults [22]. The authors of this thesis were to the best of their knowledge unable to find such dataset for React Native components written in Typescript, instead relying on the test coverage for determining the test case quality.

3.4 Temperature

Temperature is a hyperparameter used when prompting that affects the probability of which token is the next to be selected. A lower temperature results in a more deterministic result while a higher value could lead to more diverse and creative outputs [23]. Temperature does not measure the quality of unit tests, but it affects the outcome as seen in "An initial investigation of ChatGPT unit test generation" [21] and could therefore be useful when evaluating the models ability to generate test cases.

3.5 Reliability and Validity

Reliability could be affected if the React Native components have a non-representative sample. Our data will be retrieved from an existing codebase where certain coding practices can be applied. If these practises do not align with the industrial standard or differ from a readers data then the result could be affected. Another potential concern is that large language models relies on randomness, increasing the difficulty in reproducing our results. By conducting the experiment multiple times we can reduce this risk and increase reliability.

One of our validity concerns is that we are only testing on one LLM, gpt-3.5-turbo, when research like this could have been performed on several models. There is also a risk that its answers are more error prone due to its data cutoff date. But if the models follow the trends in performance seen between previous models, one could argue that its probable that the result should only improve with the use of newer models. Sampling bias is another concern, as we will only test on Softwerk AB code. The codebase provided from Softwerk AB is under a NDA-agreement and we can therefore not provide the code or test-cases generated, but only the result from it.

3.6 Ethical considerations

This thesis work does not include data related to individuals, and no code provided are linked to any political, personal, or sensitive opinions. However, the subject of this thesis, is within AI, specifically LLMs, and this introduces various ethical considerations.

The use of LLMs when testing code could give developers a false sense of trust. A developer should always examine and review code generated by LLMs, but this step could easily be overlooked or skipped. Additionally, the tests generated could compile and pass, while having poor quality. Measuring test quality is a hard task and a large part of determining if a test is good or bad, lies in the eyes of a developer. If the developer fails to recognize a fault in the code or lack of quality it could lead to errors and vulnerabilities in the software.

Another ethical consideration to be made and discussed about is the possibility that utilizing LLMs could negatively impact advancements within software testing. The model used in this thesis gpt-3.5-turbo, has a training data cutoff at September 2021 [24]. This does not only hinder developers from using it for newer technologies created after 2021, but could also promote old testing techniques and practices. Newer, better techniques may have been introduced and this delves into the next problem to consider. Using a model could remove the need for developers to discuss and further enhance testing practices, resulting in fewer advancements.

There is also little to no documented research on how the use of LLMs in a development environment psychologically affects developers. While this could potentially lead to increased efficiency, there are concerns that it may have negative effects on creativity, problem-solving skills and decreased job engagements. It could also affect the team dynamic between coworkers, shifting the interactions from collaborative problem-solving to individual interactions with automated tools.

4 Implementation

This chapter describes the implementation and technical artifacts needed to conduct the controlled experiment. In the controlled experiment 49 different testable React components written in TypeScript were retrieved from the system InfoSynk described in Section 4.4, and used to generate test files using gpt-3.5-turbo. For each component the script generated ten files, increasing the chances of generating a passing test and for each file, the API was given five possibilities of generating a compilable and passing test file. The data gathered were then used to see how well the LLM performed, to compare different temperatures and to see what issues may occur. The temperatures that was used was 0.2, 0.6 and 1, meaning that for each temperature we generated 490 test files. After the generation of tests were done for each temperature we manually ran and examined the tests. Through the setup described we were able to answer all of our research question presented in Section 1.3. The experiment setup took inspiration from previous research published in "No more manual tests?" [4], where similar artifacts were developed to retrieve their results. It was for this project deemed the best solution to retrieve quantitative data in a time efficient manner, automating larger parts of the process.

4.1 Pipeline

For this thesis a Python script was developed that finds all the tsx files inside a project, and forwards them to the gpt-3.5-turbo model with prompts discussed in Section 4.2. The script works in iterations:

1. The .tsx file is sent to the API.
2. The script receives a unit test from the API.
3. The script runs the unit test through Jest.
    - Does it compile?
    - Does the unit test pass or fail?
    - The branch coverage, line coverage, statements coverage and function coverage.
4. If the test compiles and passes.
    - A csv file with statistics for the file is generated.
    - The iteration starts over from step one and sends the tsx file again, this is done ten times per file.
5. If the test doesn't compile or fails.
    - The original prompts gets sent back together with the failed test and failing logs.
    - This is done five times if the unit test keeps failing.
    - If the test doesn't compile or pass after five times the csv file gets appended with the failing data.

As stated in the enumeration above each file is sent to the API ten times for each temperature tested. The files will be resent together with the failing unit tests and their corresponding logs a maximum of five times, if it doesn't pass by then the unit test generated will be marked as a failed attempted.

4.2 Prompts

The model gpt-3.5-turbo accepts three different types of prompts [25]:

- System
- Assistant
- User

The system role is used to tell the model what it is supposed to act like. The prompt used for this research is: *"You are a helpful software tester for React Native components using the test framework Jest"*.

The assistant prompt is used to complement a request, with each request we send this prompt as an assistant prompt: *"Always only send back the testcode and no other content"* The assistant role is also used for sending in failing unit tests with their logs, as we ask the API to try again and fix its mistakes. If the assistant has failed with generating a passing test five times in a row, we mark the test as a failure.

The user prompt is used for asking the API to do something. The first request uses the following as the user prompt: *"I need unit tests for the React Component. Only return the code and never surround it with anything. Don't test elements that is not in the component. Don't surround the testcode with "'jsx"'. Import the React Component from {path_to_react_component} (Remove the .tsx in the import). You should always write the full tests with assertions. Mock as much as possible. The React Component to be tested: react_component_text"*. The path to react component lets the API know where it should import from, whilst the react component text is the actual tsx file content.

Requests when retrying to generate a passing test: *"Fix the test file by controlling the React Component again and the failed tests, using the error logs., make sure you rewrite the whole file but only fix the error. This is the original prompt:{prompt}"*. Where prompt is the first request we made to the API.

4.3 Tools

The tools used for the implementation are Visual Studio Code, GitHub, GitLab and Git Bash. Visual Studio Code is one of the most well used code editors, used in this thesis work for development, executing of the script, and to implement the testing framework and script in the InfoSynk project. GitHub were used for version management allowing us to co-work on the development of the script. GitLab were used similarly, used for version management when developing in InfoSynk. Git Bash provides a Bash emulation, used to run Git from command lines. GitHub as well as GitLab and Git Bash are industry standards often used in the development process, allowing for an efficient workflow.

4.4 System

The system used for this research is called InfoSynk and is provided by Softwerk AB. It is private property of Softwerk AB and can't be disclosed in too much detail due to a non-disclosure agreement between Softwerk AB and us. InfoSynk is a React Native app, written in Typescript, deployed for several customers both on web, iOS and Android. It is an information platform, were customers can upload information that they want their employees to have easy access to. The application is in active development and used by many customers today. Having a active project with realistic data was the main motivation for the decision to work within InfoSynk.

4.5 Dependencies

To be able to generate unit tests for the React components a testing environment had to be established. The testing environment uses Jest as testing framework together with the compiler Babel. A testing environment has to be setup and manually modified to work with the existing InfoSynk codebase. While these settings are individual for different projects, the availability offers transparency, and could perhaps aid in a recreation of the experiment. The following dependencies are used to compile and run the unit tests:

- @babel/preset-env 7.24.5
- @babel/preset-react 7.24.1
- @babel/preset-typescript 7.24.1
- @testing-library/jest-dom 6.4.5
- @testing-library/react 15.0.6
- @testing-library/react-native 12.5.0
- @types/jest: 29.5.12
- jest 29.7.0
- jest-enviroment-jsdom 29.7.0
- ts-jest 29.1.2
- react-test-renderer 18.3.1

5 Results

This chapter presents the results from the controlled experiment. In the controlled experiment we used the codebase in InfoSynk to gather test data, retrieving 49 different testable React components. For each component, 10 tests were generated per temperature, and the temperatures tested were 0.2, 0.6 and 1. For each temperature we therefore generated 490 tests, and of these tests, only the tests that compiled and had a coverage value above zero where included. A test can therefore pass, but still not be used in our calculations below. As previously mentioned 3.5, the codebase was provided by Softwerk AB and is under an NDA, therefore no generated unit tests will be disclosed as they are private property.

In the Sections 5.1, 5.2, 5.3 we provide the specific results for the different temperatures used in the experiment. Coverage, as explained in Section 2.3.2 is provided as well as how many tests managed to compile and pass alternatively fail. The statistics also provides how many tries on an average it took to generate a passing test. Important to note is that failed tests always have an average of five tries as the LLM was given that many opportunities to make the test pass.

5.1 Temperature 0.2

Table 5.1 shows that a temperature of 0.2 resulted in 32 passing and 34 failed unit tests. The passing unit tests resulted in an average of 100% on all of the coverage metrics. It had a 13.4% chance of generating a compilable test out of the 490 tries with any coverage above zero. Out of the 66 compilable unit tests there was a 48.5% chance that the test passed and 51.5% chance that the test failed. A failing test had on average 62.14% statement-, 34.64% branch-, 36.5% function- and 61.43% line coverage.

Status	Tries	# Compiled tests	Statement %	Branch %	Function %	Lines %
Passed	**2.19**	**32**	**100**	**100**	**100**	**100**
Failed	5	34	62.14	34.64	36.50	61.43
Total	3.64	66	80.50	66.33	67.29	80.13

Table 5.1: Statistics for tests with coverage with a temperature of 0.2

5.2 Temperature 0.6

Table 5.2 shows that a temperature of 0.6 resulted in 27 passing and 32 failed unit tests. The passing unit tests resulted in an average of 100% on all of the coverage types. It had a 12.0% chance of generating a compilable test out of the 490 tries with any coverage above zero. Out of the 59 compilable unit tests there was a 45.8% chance that the test passed and a 54.2% chance that the test failed. A failing test had on average 79.99% statement-, 50.93% branch-, 61.19% function- and 79.66% line coverage.

Status	Tries	# Compiled tests	Statement %	Branch %	Function %	Lines %
Passed	**2.07**	**27**	**100**	**100**	**100**	**100**
Failed	5	32	79.99	50.93	61.19	79.66
Total	3.66	59	89.15	73.39	78.95	88.97

Table 5.2: Statistics for test with coverage with a temperature of 0.6

5.3 Temperature 1

Table 5.3 shows that a temperature of 1 resulted in 20 passing and 32 failed unit tests. The passing unit tests resulted in an average of 100% on all of the coverage types. This resulted in a 10.6% chance of generating a compilable test out of the 490 tries with any coverage above zero. Out of the 52 compilable unit tests there was a 38.5% chance that the test passed and 61.5% chance that the test failed. A failing test had on average 81.09% statement-, 83.36% branch-, 83.65% function- and 88.35% line coverage.

Status	Tries	# Compiled tests	Statement %	Branch %	Function %	Lines %
Passed	**2.25**	**20**	**100**	**100**	**100**	**100**
Failed	5	32	81.09	72.56	73.44	81.08
Total	3.94	52	88.36	83.12	83.65	88.35

Table 5.3: Statistics for test with coverage with a temperature of 1

5.4 Combined coverage

As seen in fig. 5.1, the average coverage for failing unit tests are dependant on the temperature. The temperature of 0.2 provides worst coverage on all coverage metrics, while a temperature of 1.0 provides best coverage on every coverage metric except the statement coverage.

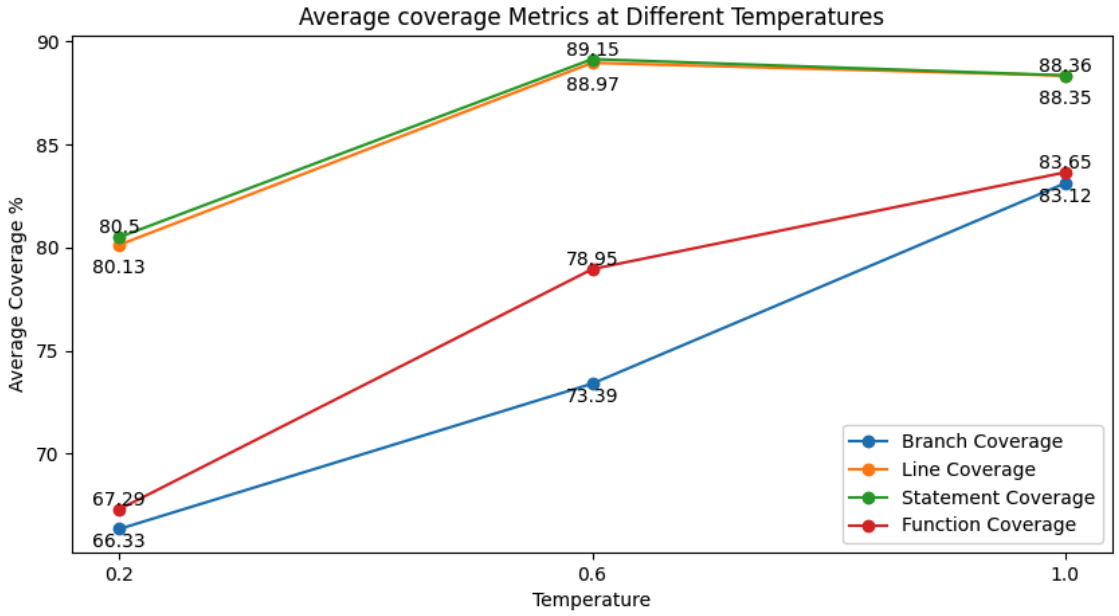

Figure 5.1: Average coverage of compilable failing tests

5.5 Test errors

There were many different scenarios where a generated test would fail to compile. Some common reasons for this is due to faulty importing, syntax errors, file transformation

errors or missing dependencies. The distribution of errors, for both compilable and non-compilable unit tests, is visualized in fig. 5.2 and fig. 5.3. The errors are explained in table 5.4.

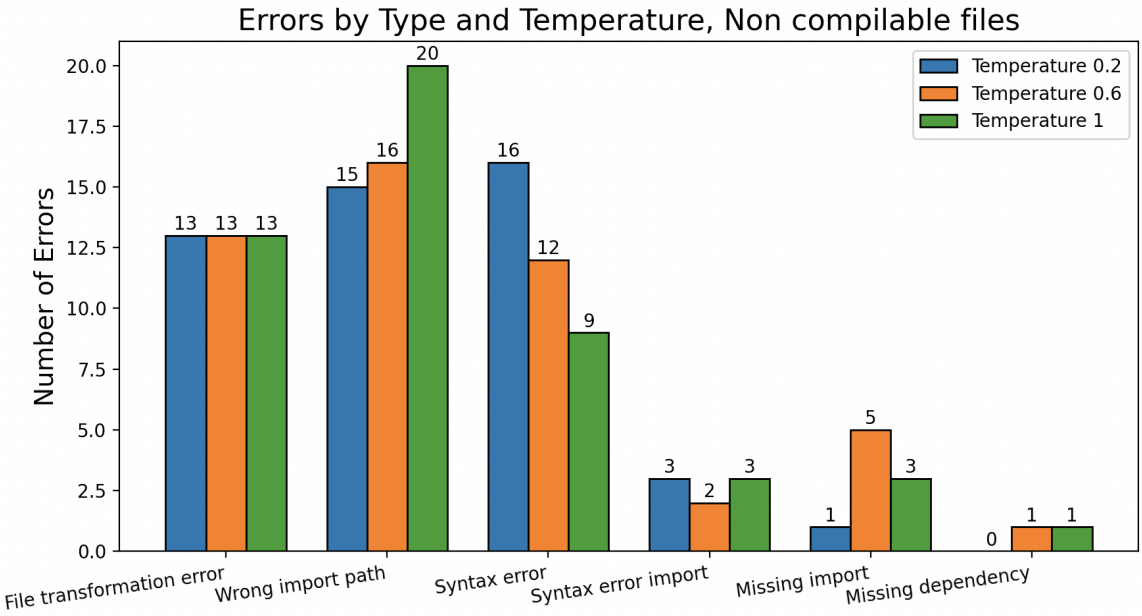

Figure 5.2: Errors on non-compilable tests

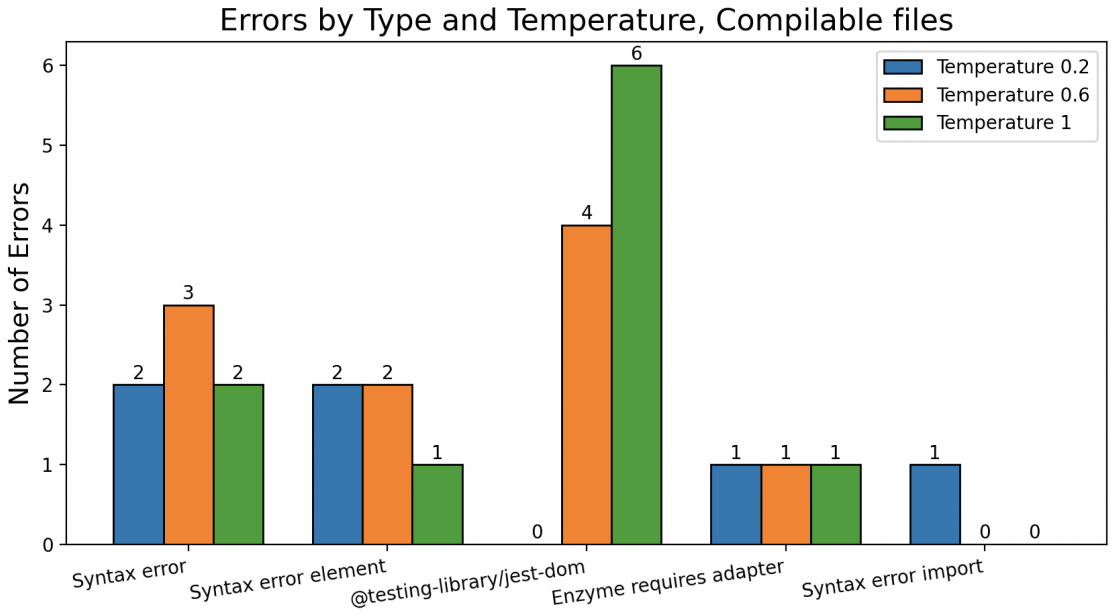

Figure 5.3: Errors on compilable failed tests

Name	Description
File transformation error	Transformation error is an error that is given when the compiler can't compile the file. This could be an issue with node_modules, js files or other types of files.
Wrong import path	Wrong import path is an error that means that the generated test file points to wrong directory or file in a import paths. This mostly happened when the generated test file was mocking a file.
Syntax error	Syntax error is an error that violates the rules of the coding language. This could be missing semicolons parentheses or other symbols. It could also be that the generated file tries to use a function that does not exist.
Syntax error import	Syntax error import was mostly the misuse of { } or trying to import modules that does not exist.
Missing import	Missing import is an error that occurs when the generated test files uses functions that isn't imported.
Missing dependency	Missing dependency is an error that occurs when a dependency is not installed that is required.
Syntax error element	Syntax error element is when the generated test tries to find an element that does not exist, for instance looking for a button with the id="testId" when it doesn't exist a button with that id.
@testing-library/jest-dom	A very common error for temperature 0.6 and 1. This occured when the generated test used functions originating from this import, but it didn't import it.
Enzyme requires adapter	Error that occurs when the generated test uses Enzyme but doesn't configure an adapter for it, which is required to use Enzyme. Enzyme is a testing utility for react applications, used for rendering of components.

Table 5.4: Description of the different errors

6 Analysis

This chapter provides an analyze of our results, chapter 6.1 analyses how different temperatures affected the tests that was generated. In chapter 6.2 an analysis of the unit tests is conducted to find common patterns in the errors, and general characteristics.

6.1 Code coverage

By analysing the statistics and test files generated from the different temperatures, we were able to observe differences when using different temperatures. The statistics displayed in Section 5 shows that a higher temperature resulted in fewer compilable unit tests. A manual analysis of the non-compilable tests was then conducted. This showed us that the tests were well written and at large mostly written in the same way, but higher temperature had a tendency of making more mistakes regarding importing, resulting in a non-compilable test. The statistics also show that the lower the temperature was the higher the chance was of generating a passing test. Notable, however is that a higher temperature resulted in a better coverage on failing unit tests, contrary to passing once.

6.2 Test errors

As shown in fig. 5.2 and fig. 5.3 there are many reasons for a test to fail. The most common issue for the non-compilable unit tests, where a wrong import path, correctable by changing the path. For the compilable unit tests, it was the missing of the import of @testing-library/jest-dom. There are many observations to be made both when analysing the individual graphs, but also when comparing them. A temperature of 0.2 had for example no instances where it forgot to import the @testing-library/jest-dom and there where also no file transformation errors or wrong import paths in the compiled failed unit tests. This suggest that a unit test can't compile when a error like this occurred, compared to syntax error which is found in both graphs, not always resulting in a non-compilable test. To retrieve the data in fig. 5.2 and fig. 5.3 the authors manually ran the tests and observed the error messages as well as the code. Something noteworthy is that according to the authors, the tests were at large well written, with good descriptions of what was tested, and well structured with the AAA pattern described in Section 2.4.1.

7 Discussion

This chapter evaluates and discusses the research questions relative to the results provided through the controlled experiments.

RQ1: Can LLMs be used to generate comprehensive and accurate unit tests for React Native components written in TypeScript?

A controlled experiment were in this thesis conducted to answer if an LLM is able to generate unit tests, how well it is able to generate unit tests, what quality the unit tests have and what problems occur when generating unit tests. Such experiment had previously been performed on other coding languages and frameworks but not on React Native components written in TypeScript.

This study was to the best of our knowledge the first performed within the field on React Native components, making direct comparable data unavailable. But through the controlled experiments we where able to generate data, able to answer the previous stated questions. We can through the quantitative data displayed in Section 5 conclude that it indeed was able to generate comprehensive and accurate unit tests. The experiment was performed three separate times, one for each temperature, generating a total of 1490 test files. The content of the test files showed similar results, regarding temperature, indicating that the same result would most likely be achieved if performed again.

RQ2: What impact do different temperatures have on the generated unit tests?

The first controlled experiments were performed with different temperatures, and the result of these experiments tells us that different temperatures do impact LLMs ability to generate unit tests. The hyperparameter temperature, as explained in Section 3.4 affects the probability for the next token to be chosen. A lower temperature makes the LLM more deterministic and the opposite is true for higher temperatures. In the experiment, the lowest temperature of 0.2 performed best, achieving the most amount of compilable unit tests, needing the fewest amount of tries and with the highest possibility of generating a passing test. The effect of different temperatures have been documented in previous research, but as their prompting techniques and code language differs a direct comparison can't be made.

RQ3: What type of problems are detected when generating unit tests using LLMs?

The second part of the experiment was performed by manually running 48 failed unit tests on three different temperatures, both compilable and non-compilable, documenting the different errors that occurred. There are many conclusions that can be made from the data retrieved. The experiment answers the research question well, clearly indicating the main cause for the non-compiling unit tests inability to compile. The compilable unit tests had a higher spread when comparing the frequency of the different problems, potentially due to the lower amount of unit tests. The two graphs also provide a good indication on how the errors differentiate between the compilable and non-compilable tests.

While the findings to all three research questions are promising, suggesting that LLMs can effectively generate unit tests for React Native components written in TypeScript, the results should be generalized with caution. Further studies with varied components, LLMs, and environments are recommended to reinforce the confidence in these conclusions.

8 Conclusions and Future Work

In this chapter we present our final conclusion from this research and work, we also highlight future possibilities.

8.1 Conclusion

To answer the research questions, we created a Python script capable of generating test files for React Native components written in TypeScript by the use of the model gpt-3.5-turbo. The script and API clearly shows acceptable and interesting results, managing to generate well written and working tests for the React components. The process of generating unit tests was also conducted using various temperatures to explore the potential effects of different settings.

The results clearly shows that LLMs can be used to generate comprehensive and accurate tests for React Native TypeScript components. All tests generated did not compile, but even if the test didn't compile they still showed great potential and most often failing due to minor mistakes. Most of the mistakes were easily correctable, and with an newer model, or a model that can be fine-tuned on the specific codebase, many of the problems could probably be avoided. From the experiment the results also show that prompting with different temperatures affects the results. The higher the temperature was, the more importing errors was made. However, the tests were written in an accurate and comprehensive way no matter what temperature was used, and higher temperatures actually showed better coverage for failed compilable tests when compared to the lower temperature. From this we can deduce that different temperatures do have a large impact, and from the results presented in this paper, a lower temperature is recommended as it had a better ability to generate compilable unit tests.

The results also highlights the different errors made by the gpt-3.5-turbo model, illustrated in fig. 5.2 and fig. 5.3. Most issues were due to file transformation errors, wrong import path or syntax errors explained in table 5.4. These errors with the right dependencies and setup would most likely be less frequent. As analysed in Section 7 and discussed in Section 8 most errors occurred due to imports pointing to the wrong folder or missing dependencies. This could be counteracted by implementing the script from start, making it work better with the dependencies of the project. The ability to ensure that the test files were generated next to the file that was under testing would emerge, probably decreasing importing errors.

The thesis shows that LLMs can be useful in generating unit tests for React Native components, possibly increasing the workflow efficiency within a project. Even if the tests were not always compiling and passing they could still be valuable to the industry. By generating tests and then correcting potential issues it can reduce the development time significantly. More time can be used on perfecting the unit tests, resulting in higher quality, leaving more time to do extensive testing. If a company were to use an enterprise model or train their own model on their codebase, improvements in test case quality is highly possible. The codebase used within this thesis is an active and newly developed product, arguing for the test data to be realistic and the result therefore highly relevant to the industry.

The knowledge gap explained in Section 1.1 and Section 1.3 is with this thesis covered. By the results we can observe that time can be saved by generating unit tests with the use of an LLM. It can help save time for developers who might normally skip the testing process or doesn't have the required knowledge of writing good test cases. Previous

research had been done for other frameworks and languages, but to the best of our knowledge not for React Native TypeScript. Through this experiment and thesis, we show that the use of LLMs is a valid and possible option.

8.2 Future work

There are multiple directions available moving this work forward. The easiest change with the highest possibility of receiving a better result, would in our opinion be the testing of other large language models, such as gpt-4. If large language models follows the trend of improvements seen between previous models, one could assume future models would perform better. This was unavailable for us due to the high cost of gpt-4. The relationship between the effect of temperature and other models would also be of great interest. The second area to further investigate is prompt engineering. For this thesis work, very little time was spent on experimenting with different prompts. Different prompts can have a great effect on LLMs and experimenting with different prompts could be of high value. It could also be of great value to test LLMs on a open-source library, or another codebase. Retrieving all components from one codebase causes limitation and potentially bias, but there is also another potential advantage. If tested on a codebase with existing unit tests, the tests could be used to train the model, enhancing its generative abilities. The model could also be trained on the whole project, making it more optimized to generate tests for said project.

Another path available is to further develop our test generation script, further developing the artifact into a complete product. A tool for automatically generating unit tests for React Native components could give much value. It's a well used framework and the use of such product could cut down the development time and thereby cost of projects. The reconstruction into a tool could include giving a private large language model full access to the source code, removing the chance to generate unit tests with faulty import paths, probably improving its ability to generate compiling unit tests.

References

[1] J. Pan, "Software Testing," *https://citeseerx.ist.psu.edu/*, September 2006, Accessed: 03 Apr. 2024. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=28abbfdcd695f6ffc18c5041f8208dcfc8810aaf

[2] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024, Accessed: 03 Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10329992

[3] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai : A comparative performance analysis of autogeneration tools," 2024, Accessed: 03 Apr. 2024. [Online]. Available: https://arxiv.org/abs/2312.10622

[4] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," 2023, Accessed: 03 Apr. 2024. [Online]. Available: https://arxiv.org/abs/2305.04207

[5] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, pp. 1–27, 2024, Accessed: 03 Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10440574

[6] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211, Accessed: 10 May 2024. [Online]. Available: https://ieeexplore.ieee.org/document/6982627

[7] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically testing javascript apis with asynchronous callbacks," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1494–1505, Accessed: 14 May 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9793885

[8] A. Walgude and S. Natarajan, "World quality report," Accessed: 10 Apr. 2024. [Online]. Available: https://www.capgemini.com/es-es/wp-content/uploads/sites/16/2019/10/World-Quality-Report-2019-20.pdf

[9] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with gui test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021, Accessed: 12 Apr. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000963

[10] M. Shanahan, "Talking about large language models," 2023, Accessed: 04 May 2024. [Online]. Available: https://arxiv.org/abs/2212.03551

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023, Accessed: 05 May 2024. [Online]. Available: https://arxiv.org/abs/1706.03762

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016, Accessed: 01 May 2024. [Online]. Available: https://arxiv.org/abs/1409.0473

[13] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," 2017, Accessed: 30 Apr. 2024. [Online]. Available: https://arxiv.org/abs/1703.03906

[14] G. Yenduri, M. Ramalingam, G. C. Selvi, Y. Supriya, G. Srivastava, P. K. R. Maddikunta, G. D. Raj, R. H. Jhaveri, B. Prabadevi, W. Wang, A. V. Vasilakos, and T. R. Gadekallu, "Gpt (generative pre-trained transformer)—a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions," *IEEE Access*, vol. 12, pp. 54 608–54 649, 2024, Accessed: 05 May 2024. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10500411

[15] Github. Programming languages. Accessed: 21 Apr. 2024. [Online]. Available: https://innovationgraph.github.com/global-metrics/programming-languages

[16] D. Choi, *Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL*. Packt Publishing, Dec. 2020, Accessed: 10 Apr. 2024. [Online]. Available: https://books.google.se/books?hl=sv&lr=&id=uUMQEAAAQBAJ&oi=fnd&pg=PP1&dq=React+Native+TypeScript&ots=KZC_By5qlw&sig=HpdaAH_glGH1UMlM666BFsnd0zE&redir_esc=y#v=onepage&q=React%20Native%20TypeScript&f=false

[17] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*, 2020.

[18] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006, Accessed: 25 Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/6982627

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin, Heidelberg, June, 16 2012, Accessed: 05 Apr. 2024. [Online]. Available: https://link.springer.com/book/10.1007/978-3-642-29044-2

[20] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 955–963. [Online]. Available: https://doi.org/10.1145/3338906.3340459

[21] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, ser. SAST '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 15–24, Accessed: 25 Apr. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3624032.3624035

[22] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, pp. 402–411, Accessed: 25 Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/1553583

[23] M. Renze and E. Guven, "The effect of sampling temperature on problem solving in large language models," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.05201

[24] Microsoft. (2024, May) *Azure OpenAI Service models* [Online]. Accessed: 10 May 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/aiservices/openai/concepts/models#gpt-35-turbo-modelavailability

[25] "Openai api promptengineering," Accessed: 25 Apr. 2024. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering