Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--25/021--SE

Evaluating Generative AI's Applicability in Software Testing

– User study and quality evaluation of unit test generation

Evaluering av generativ AI:s tillämpbarhet inom mjukvarutestning
- Användarstudie och kvalitetsevaluering inom generering av enhetstester

Sagar Batra
Oskar Danielsson

Supervisor : Xin Sun
Examiner : Dániel Varró

External supervisor : Azeem Ahmad

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

This thesis explores the current capabilities and perceptions of generative AI tools for software testing in an industrial environment, focusing on unit test generation. Despite the potential for generative AI to automate test generation and improve efficiency, the study reveals challenges related to output quality, compilability, and practical integration barriers. Using a commercial coding assistant, we generate unit tests for an industrial code base. We then evaluate the generated tests, revealing a compilation rate of 36.09% and emphasizing compilability as the primary obstacle in generating high-quality tests. To reveal the perceived benefits and challenges of using generative AI in software testing, we conduct interviews with engineers to gather qualitative data and perform a thematic analysis. The analysis reveals perceived benefits such as time savings, efficiency improvements, and organizational support and encouragement. It also highlights challenges related to security and data confidentiality concerns, validation requirements, and output quality and reliability. While engineers recognize the theoretical potential of generative AI tools in unit test generation, the adoption of such tools in industry remains limited, with some viewing it as a misapplication of the technology. This research contributes to more informed decision-making for companies considering integrating generative AI tools into their software testing workflows.

Acknowledgments

We would like to express our gratitude to our supervisor Azeem Ahmad at the company we conducted our thesis at, for guiding and supporting us throughout our work. In addition, we would like to thank our supervisor Xin Sun at Linköping University for providing us with valuable feedback and support throughout the whole period. We greatly appreciate the involvement of our examiner Dániel Varró, who held us to high standards and kindly stepped in to take on the duty as our examiner mid-term. Lastly, we extend our gratitude to our friends and families, who have been constantly supporting us throughout the period.

Contents

List of Figures

List of Tables

1 Introduction

This chapter includes a motivation that explains the significance and relevance of the thesis. Furthermore, it presents the research questions, contributions and delimitations before finally outlining the structure of the thesis' contents.

1.1 Motivation

Software testing is an important part of software engineering, ensuring that user requirements are met and allowing for preemptive detection of crticial bugs which, if left undetected, could cause issues such as system crashes or security vulnerabilities. While preventing such issues saves on both time and money, software testing tends to require plenty of resources [51], often accounting for at least half of total project costs [39]. Manually writing tests requires significant effort and is time-consuming and tedious. This makes exploring the possibility of automating test generation important. Previous research has already shown that automatic test generation tools are able to generate tests that are better than manually written ones, according to metrics such as *coverage* and *mutation score* [45]. This highlights the potential of automating test generation, while maintaining a high-quality test suite. Unit testing is a type of software testing that aims to test the smallest individual software components called *units*. Unit tests are the foundation for other types of test such as integration tests, which instead test the integration of multiple such units. Due to the relative simplicity of unit tests in comparison to other software tests, there is high potential for automating their generation.

In recent years, large language models (LLMs) have revolutionized the field of artificial intelligence (AI), demonstrating remarkable capabilities across various domains [31]. Despite the popularity of LLMs, leveraging them for domain-specific tasks, such as software testing, remains a complex challenge requiring customization and adaptation [34]. While LLMs are more commonly famous for their ability to process natural language, recent studies have shown impressive improvements on code generation as well. The HumanEval leaderboard[1] shows how the 2023 Palm 8B model [11] scored 3.6% on the Pass@1 metrics, in comparison to the 2024 LDB model [52] scoring 95.1% [25]. With such rapid improvements in key metrics, exploring the practical implications and capabilities becomes of high importance for companies wanting to keep up technologically.

[1] https://paperswithcode.com/sota/code-generation-on-humaneval

This thesis was carried out at a telecommunications company, which was interested in adopting a coding assistant utilizing generative AI for unit test generation. The company, recognizing the growing number of development tools in their ecosystem, preferred to explore extending their newly integrated coding assistant to specific tasks like unit test generation rather than developing another specialized tool. To quantify the strength of such a tool for unit test generation, this thesis evaluates the coding assistant by measuring the quality of the generated unit tests in regards to four quality metrics: compilation success rate, Levenshtein similarity, cosine similarity, and Jaccard similarity.

Compilation success rate measures compilability by executing the generated tests in the company's codebase. Compilability acts as a fundamental prerequisite for test quality, as non-compiling tests cannot be executed or validated further. Edit distance [32] has been shown to effectively measure similarity between test cases [28]. As such, Levenshtein similarity, a normalized variant of edit distance, is used in this thesis. Furthermore, we use cosine similarity to benchmark our results against comparable evaluations of generated tests previously conducted within the company. Levenshtein and cosine similarity can be skewed when handling repeating code. As such, we use Jaccard similarity which disregards repeated keywords and measures the overlap of unique terms.

The company was also curious about how the people working in software testing feel about using generative AI in their work. As the capabilities of generative AI improve, and the barrier of entry decreases, more companies will be interested in exploring the adoption of generative AI tools and how it will affect their workforce. A main part of this thesis consequently includes a user study which explores the perceived benefits and challenges among engineers in using generative AI for software testing tasks.

1.2 Aim

This thesis aims to explore the viability of using commercial coding assistants for generating unit tests in an industrial context. Specifically, the objective is to evaluate one such coding assistant in terms of the quality of generated unit tests with the goal of gaining insights about its capabilities for this task in an industrial context. Furthermore, the thesis aims to provide insights from people who work with testing about how they perceive using generative AI in their work, in terms of benefits and challenges, in order to guide teams that are looking to implement such tools into their workflow.

1.3 Research questions

The thesis is guided by the following research questions:

RQ1: How is the quality of unit tests generated by a commercial generative AI tool in an industrial context?

RQ2: What are the benefits and challenges of adopting generative AI tools for software testing as perceived by engineers?

RQ1 investigates the quality of tests generated by a coding assistant, and focuses specifically on unit tests. We evaluated the quality of unit tests in terms of compilation success rate, cosine similarity, Jaccard similarity, and Levenshtein similarity. We chose to investigate the compilation success rate because it describes the likelihood of a generated test not causing any errors. The three similarity metrics were chosen due to their diverse and unique strengths in measuring text similarity using different aspects of the text (see Section 2.3). This research question provides insights about the viability of generating unit tests using a commercial coding assistant.

RQ2 instead explores perceptions about using generative AI tools for software testing in general. We performed a user study by conducting interviews with engineers at a telecommunications company. This research question aims to provide insights into the adoption of generative AI tools for software testing in order to help companies make more informed adoption decisions.

1.4 Contributions

The main contributions of the thesis are the following:

1. We evaluate the quality of unit tests generated by a commercial coding assistant, and find relevant factors for determining the quality of the generated unit tests.
2. We perform a user study about perceptions of using generative AI for software testing tasks. We identify and present the perceived benefits and challenges found through a thematic analysis.

Added value The company can use the analysis of the commercial coding assistant to get an understanding about its applicability for unit test generation for their specific code base. Furthermore, our thematic analysis of the user study showcases the most common perceived benefits and challenges in using generative AI for software testing tasks. Company executives can use this knowledge to identify possibilities and prepare for challenges before even deciding to adopt such technologies in an era where keeping up with AI technologies is difficult but important for maintaining productivity and competitiveness.

1.5 Delimitations

Due to confidentiality, neither the company nor the coding assistant is mentioned by name. Furthermore, information such as the exact size of the dataset, characteristics of the code base and identifiable information about the interview participants is excluded. Additionally, some information, especially related to the coding assistant, is unknown even to us and therefore not included in this work. Together, these delimitations make the experiment difficult to reproduce, especially the methodology used to answer RQ1.

1.6 Structure

This thesis is divided into a total of six chapters. Chapter 2 introduces the relevant background information for understanding the subsequent chapters, and ends with a section about previous related work. Chapter 3 describes the methodology used to investigate the research questions, first describing the performance evaluation for RQ1, before moving on to the user study for RQ2. The results of these investigations are presented in Chapter 4. Chapter 5 consists of a discussion about the results and methodology as well as societal and ethical implications of using generative AI for software testing. Finally, Chapter 6 concludes the thesis by providing concise takeaways for the research questions.

2 Background and Related Work

This chapter provides an overview of the key concepts and technologies that form the foundation of the thesis. Section 2.1 outlines the principles of software testing and its various levels. Section 2.2 focuses on unit testing using the GoogleTest framework[2] which the code base we use in our study utilizes. Section 2.3 introduces the evaluation metrics used to assess the efficacy of generated unit tests, and their similarity to the the company's existing tests. Section 2.4 discusses large language models, emphasizing their architecture and application in generative AI tasks. Section 2.5 explores retrieval-augmented generation, highlighting how it enables context-awareness in large code bases. Section 2.6 introduces coding assistants, and aims to give an understanding of how it combines large language models with retrieval-augmented generation to assist programmers. Section 2.7 presents three well-established theoretical frameworks that explain what drives adoption of new technologies. Finally, Section 2.8 examines related work in the fields of unit test generation and perceptions about generative AI.

2.1 Software testing

Software testing is an essential phase in software development, focused on validating the quality and functionality of software products. Myers et al. describe software testing as "a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended" [33]. Testing spans various levels, including unit testing, integration testing, system testing, and acceptance testing as can be seen in Figure 2.1. Each level addresses different aspects of the software and serves a distinct purpose.

The lowest level is *unit testing*, or sometimes referred to as component testing, which aims to test the smallest testable components of the software, such as classes or functions [24]. Unit testing serves to identify errors at an early stage, where the cost of correcting these errors is much smaller, and the probability of correcting them correctly is higher, compared to later stages of development [33]. By ensuring individual components perform as expected, unit testing helps build a solid foundation for the subsequent stages of testing.

[2]https://github.com/google/googletest

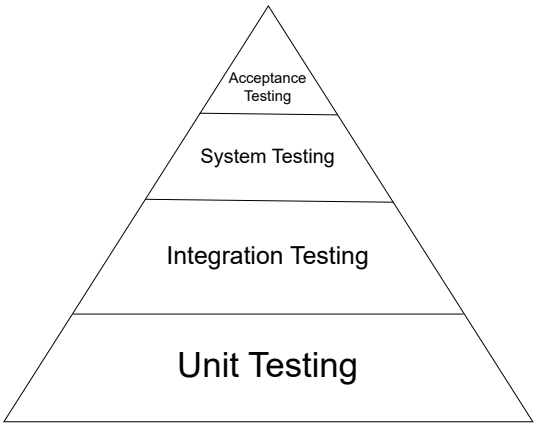


Figure 2.1: Overview of software testing levels

Above unit testing is *integration testing*, which is the process of testing and combining components (or units) so that their interactions are evaluated. Integration testing focuses on detecting interface defects and interaction errors between integrated units or modules [24]. It ensures that different parts of the software work together, and it often uncovers issues that may not be apparent during unit testing.

System testing comes next in the hierarchy, with the purpose of testing the software as a complete system. This level of testing evaluates the end-to-end functionality and behavior of the entire application, ensuring its compliance with the specified requirements [24]. System testing is crucial for verifying that the software performs correctly in the intended environment and under expected conditions.

Finally, *acceptance testing* is conducted to establish whether the software satisfies its acceptance criteria, enabling the customer to determine whether to accept the system. This level of testing is often performed by the end-users, customers, or other authorized entities [24]. Acceptance testing often serves as the final validation step to ensure that the system meets the specified requirements and expectations.

Dividing software testing into layers helps identify issues progressively, reducing the risk of costly fixes and rework. By capturing defects early on, testing minimizes the likelihood of problems surfacing in production, thereby enhancing the overall quality and reliability of the software product.

2.2 Unit testing and the GoogleTest framework

While unit testing was previously introduced as a foundational level of software testing, this section provides a more practical and detailed view of how unit testing is applied within the context of this thesis, specifically using the *GoogleTest*[2] framework for C++.

In general, a *unit test* is a small program that verifies the correctness of an isolated software component which is commonly referred to as the *method under test* [5][47], hereafter referred to as the *MUT*. Unit tests are designed to test individual functions or classes in isolation, ensuring that they behave correctly under various conditions.

The software system analyzed in this thesis is implemented in the C++ programming language and uses the GoogleTest framework for unit testing. A unit test in GoogleTest is written using one of three available macros: `TEST`, `TEST_P` and `TEST_F`. The `TEST` macro is used for simple test cases that do not require a shared state. For test cases that need to run the same logic across multiple inputs, the `TEST_P` macro enables testing with different parameters. Finally, the `TEST_F` macro is used when tests require access to a predefined set

[2]https://github.com/google/googletest

of objects or data shared between multiple tests. These tests use a fixture, which is a class that encapsulates the common setup logic between the tests. To verify that the behavior of the MUT test matches expectations, GoogleTest provides a set of assertion macros such as `EXPECT_EQ`, `EXPECT_NE`, `ASSERT_TRUE` and `ASSERT_FALSE` among others. These macros compare actual values to expected ones and either report failures or abort the test depending on the severity of the assertion used.

Within the software system analyzed in this thesis, the vast majority of test cases are written using the `TEST_F` macro. A small number of tests use the `TEST` or `TEST_P` macros, but these are infrequent and have therefore been excluded from this work.

To illustrate the use of GoogleTest with fixtures, consider Figure 2.2 depicting a simple `Counter` class that increments an internal count, a fixture setting up an environmental state, and a unit test declared with the `TEST_F` macro that validates the increment function.

```cpp
// Simple Counter class
class Counter {
public:
    Counter() : count_(0) {}

    void Increment() { ++count_; }

    void Reset() { count_ = 0; }

    int GetValue() const { return count_; }
private:
    int count_;
}
```

```cpp
// Fixture
class CounterTest : public testing::Test {
protected:
    Counter counter_;
    void SetUp() override {
        counter_.Increment();}  // Initialize to 1
};

// Test
TEST_F(CounterTest, Increment) {
    EXPECT_EQ(counter_.GetValue(), 1);  // Initial state
    counter_.Increment();
    EXPECT_EQ(counter_.GetValue(), 2);  // After increment
}
```

Figure 2.2: An example of the GoogleTest framework.

In this example:

- `Increment()` is the MUT.
- `CounterTest` is a fixture that sets up a new `Counter` object where the counter is initialized to 1.
- `TEST_F` is used to define a unit test that invokes the MUT and verifies the result with assertions.

2.3 Evaluation metrics

To measure the quality of the generated tests, their compilability can be determined by executing them. However, quality can also be measured by measuring similarity between generated tests and pre-existing tests for the same MUT. The pre-existing tests are considered **gold standard** equivalents, and a higher similarity score thus indicates that the generated test more closely resembles its gold standard equivalent in terms of features that the corresponding metric measures. It also indicates that the unit tests have higher quality, since it means that the tests follow the company's testing standards to a higher degree and thus are adapted to the company's code base. With a higher similarity score, these tests are also more likely to test the same aspects of the MUT as their gold standard equivalents.

While similarity metrics used for plagiarism detection struggle with capturing similarity between test cases, edit distance [32] shows more promising results [28]. Consequently, this thesis makes use of Levenshtein similarity which is a normalized variant of edit distance.

Previous evaluations of unit test generation at the company used cosine similarity, another text-based similarity metric, due to its focus on entire tokens as opposed to single characters.

As a consequence, the company requested the use of the cosine similarity metric in this thesis, in order to be able to compare this approach to previous ones in terms of similarity.

Although cosine similarity and Levenshtein similarity can capture similarity from different perspectives, both metrics can be skewed when dealing with repeating code. To combat this, Jaccard similarity is chosen as the final similarity metric, as it provides a measurement of the overlap of unique terms, thus disregarding any repeating keywords.

Below are descriptions of what definitions we use for these metrics, what the metrics actually measure, and a motivation behind the choice of each metric.

Compilation success rate

The compilation success rate describes the ratio of tests in a set that are compilable, as described by Equation 2.1. This metric is included because it describes the rate of generated tests that do not cause any compilation error when executed. Compilability is an important quality metric because it is not viable to add a generated test that is not compilable to the code base without additional modification. A high compilation success rate can be interpreted as there being a higher likelihood for a given test to compile without any errors.

$$\text{Compilation success rate} = \frac{\text{Number of compilable tests in the set}}{\text{Number of total tests in the set}} \tag{2.1}$$

Cosine similarity

Cosine similarity [43] is a metric used to calculate the angular similarity between two vectors. It does not take the magnitude of the vectors into account, instead relying on the angle θ between the vectors. Applying the cosine function on θ, as described by Equation 2.2, gives a similarity score between the two vectors $\mathbf{A}$ and $\mathbf{B}$ in the range [-1, 1], where a higher similarity score indicates that the vectors point in a similar direction.

$$\text{CosineSimilarity}(\mathbf{A}, \mathbf{B}) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|} \tag{2.2}$$

However, we are not interested in the similarity between two vectors, but rather the similarity between two unit tests: the generated unit test ut_{gen} and its gold standard equivalent ut_{gold}. To solve this problem, we vectorize the two unit tests. This is done by first collecting a list L of length n, consisting of all unique terms from both of the unit tests. Then both unit tests can be described using vectors V_{gen} and V_{gold}, each of length n, where $V_{gen,i}$ is the frequency of the term L_i in ut_{gen} and $V_{gold,i}$ is the frequency of the term L_i in ut_{gold}.

Rationale in a software testing context: Cosine similarity was chosen as a metric because it describes the term frequency in the code. To what extent the generated tests tend to include terms in a similar frequency as their gold standard equivalents do, is an important metric for determining quality, as a high cosine similarity is an indication that the generated test is using the right keywords in the right proportions.

Jaccard similarity

The Jaccard similarity [17] measures the rate of shared terms between two sets A and B, as described by Equation 2.3. This metric measures the presence and absence of terms, without considering their importance or frequency, in contrast to cosine similarity. To calculate the Jaccard similarity, the input is two sets A and B, which in this case are the sets of unique terms in the generated and golden unit test respectively.

$$\text{JaccardSimilarity}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

Algorithm 1: Calculation of Levenshtein distance

Data: Strings a and b
Result: Levenshtein distance $D(|a|, |b|)$
Function `LevenshteinDistance(a, b)`**:**
  $n \leftarrow |a|$; $m \leftarrow |b|$;
  Initialize matrix $D[0 \dots n, 0 \dots m]$;
  for $i \leftarrow 0$ **to** n **do**
    $D[i, 0] \leftarrow i$;
  end
  for $j \leftarrow 0$ **to** m **do**
    $D[0, j] \leftarrow j$;
  end
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** m **do**
      if $a[i] = b[j]$ **then**
        $cost \leftarrow 0$;
      else
        $cost \leftarrow 1$;
      end
      $D[i, j] \leftarrow \min\left(D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + cost\right)$;
    end
  end
  return $D[n, m]$;

Rationale in a software testing context: While this metric has some overlap with cosine similarity, it is unique in placing focus on presence and absence of terms, which is another indicator of quality, as the presence of key terms is necessary to properly test the MUT. A high Jaccard similarity indicates that the generated test shares a larger portion of the unique keywords with the gold standard equivalent.

Levenshtein similarity

Levenshtein distance [21] is a metric describing how many single-characters edits are needed to change one string into another. We define Levenshtein similarity as a normalized and inverted version of the Levenshtein distance (see Algorithm 1) in order to transform it onto the interval [0, 1]. It is calculated as shown in Equation 2.4 and is a measure of the similarity between two strings A and B on a single-character level.

$$\text{LevenshteinSimilarity}(A, B) = 1 - \frac{\text{Levenshtein distance}(A, B)}{\max(|A|, |B|)} \tag{2.4}$$

Rationale in a software testing context: While cosine similarity and Jaccard similarity work on a term level, Levenshtein similarity works on a character level, taking into account not only slight differences between terms, but also the order of the string. Thereby it provides a different type of text-based similarity score to complement the previous two. A high Levenshtein similarity can be interpreted as the generated unit test having similar characters, words, and overall structure as its gold standard equivalent.

2.4 Large language models

Generative AI is a subfield of artificial intelligence, as shown in Figure 2.3, consisting of models that are able to produce new data based on patterns learned from the training data.

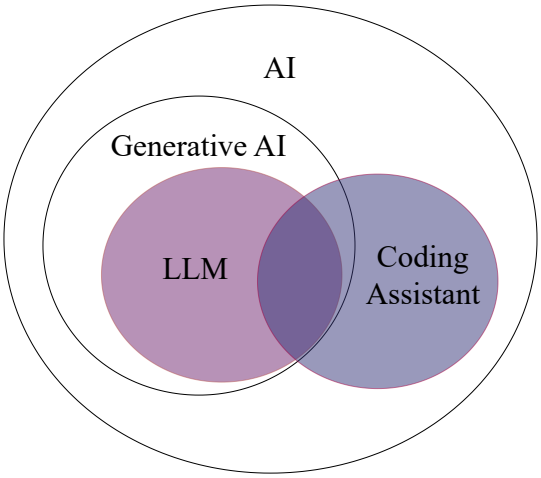


Figure 2.3: An illustration of the overlap between the relevant generative AI topics

One example of generative AI is the large language model (LLM), which is a type of machine learning model that is commonly used for text generation. The model is commonly trained using large corpora of text, letting the model learn patterns in natural language to form an understanding of how language works. After being trained, a large language model can be given an input string (*prompt*) and is able to predict a continuation using its learned understanding of natural language. This behaviour is applicable to a multitude of tasks such as language modeling, summarization, and translation [38].

Vaswani et al. [48] introduced the Transformer in 2017, a network architecture for LLMs relying on a mechanism known as *self-attention*. This mechanism allows a model to weigh and integrate different parts of an input sequence to capture contextual relationships effectively. The transformer architecture described by Vaswani et al. consists of encoder and decoder blocks. Since then, the decoder-only type has emerged as the most popular transformer type in LLMs [36]. Below is a description of this decoder-only model type, with bolded names representing the model parts that are displayed in Figure 2.4

All input strings to an LLM are first passed through a tokenizer that breaks down each string into subwords (*tokens*). The tokens are then converted to corresponding numbers called *token IDs* using a look-up table vocabulary V. The resulting sequence of token IDs of size n is then passed through the layers of the LLM as detailed in the following sections.

2.4.1 Embedding layer

The **Embedding** layer consists of an embedding matrix $\mathbf{E}$ of dimensions $|V| \times d$ where $|V|$ is the size of the vocabulary and d is the dimensionality of the embeddings [48]. The main purpose of this layer is to convert the token IDs into vector representations in order to capture semantic meaning, allowing the model to perform operations on natural language context. The embedding matrix can be thought of as a look-up table with the token ID t_i used as a key to find the corresponding row of length d, representing a vector. The operation is described with Equation 2.5. Before moving on to the decoder blocks, a **positional encoding** is added in order to include information about the order of the tokens [48].

$$\mathbf{e}_i = \mathbf{E} \times t_i \tag{2.5}$$

2.4.2 Decoder block

After the Embedding layer and positional encoding, the model has N sequential **decoder blocks** [16]. The embedded and positionally encoded input information is passed through

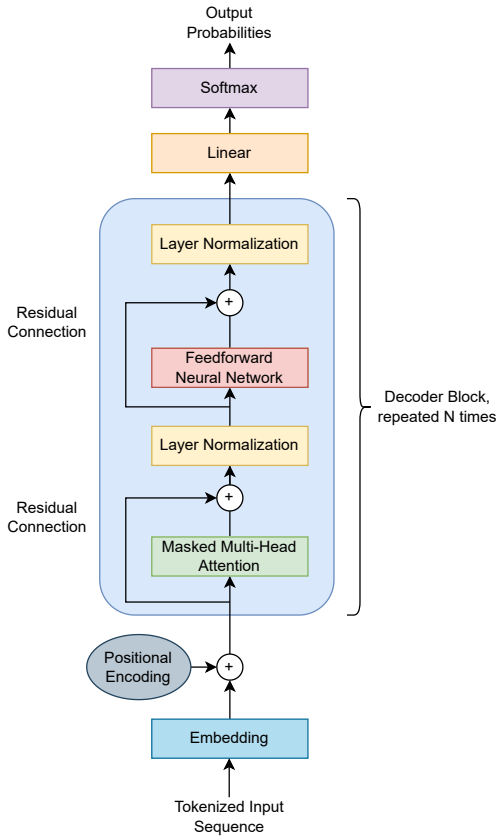


Figure 2.4: Decoder-only transformer architecture. Figure inspired by Radford et al. [37]

these blocks sequentially. The structure of a decoder block varies between models, but there are three central components: **masked multi-head attention**, **feedforward neural network** and **layer normalization**.

The *attention* part of masked multi-head attention refers to the mechanism introduced by Vaswani et al. [48] mentioned above. As the name suggests, it lets the model give different amounts of importance to the different tokens in the sequence. The *multi-head* part refers to a structure where the model has different attention *heads*, collections of parameters that learn diverse aspects of the relationships between tokens.

To prevent the model from attending to tokens later in the sequence, a *mask* M is added which alters the scores for future tokens in order to skip them. Equation 2.6 describes how attention is calculated for each head h_i. The matrix Q represents the word for which the attention score is calculated, and d_k is the dimensionality of keys in each attention head. The matrices K and V represent the words that are being attended to and the information that is being aggregated, respectively [16]. Equation 2.7 describes how the heads are combined into a single output. The task of the learned output projection matrix W^O is to combine the heads in a meaningful way, as the heads might tend to different parts of the input sequence [16].

$$h_i = \text{MaskedAttention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M\right) V_i \tag{2.6}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \tag{2.7}$$

The feedforward neural network typically consists of two linear layers [48], the first layer projecting the input to a space of higher dimensionality and the second layer projecting it back to the original dimensionality. Between the two layers there is a non-linear activation

function, which allows the model to learn more complex patterns and relationships. Equation 2.8 describes the operation, assuming the usage of the commonly used ReLU activation function [48]. W_1 and W_2 are learnable weight matrices, and b_1 and b_2 are biases.

$$\text{FeedForwardNetwork}(X) = \text{ReLU}(XW_1 + b_1)W_2 + b_2 \tag{2.8}$$

The decoder block also typically includes so called residual connections [48] for both the masked multi-head attention and for the feedforward neural network [16]. These connections help the model to keep the context from the input and tend to be followed by a layer normalization layer. For each data point, these layers compute the mean and variance across all features, and subsequently normalize the activations. The layer normalization layer includes parameters for scaling and shifting, allowing the model to reduce the normalization if doing so could improve the performance [3]. The aim of this layer is to stabilize the training and to reduce the sensitivity to the initial weights.

2.4.3 Generating text

This decoder output of size $d \times n$ is passed through a **linear** layer and transformed to a size of $d \times |V|$, before the **softmax** function [48] seen in Equation 2.9 is applied. In the resulting matrix of size $n \times |V|$, each row represents the probability distribution over the vocabulary for the corresponding token position in the input sequence. The next token in the sequence can be generated by either simply selecting the token with highest probability, or by sampling from the probabilities. This is repeated until an end-of-sequence token has been generated, or until a limit of maximum sequence length has been reached.

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.9}$$

2.5 Retrieval-augmented generation

Retrieval augmented generation (RAG) is a hybrid approach introduced by Lewis et al. [29] which combines non-parametric memory and parametric memory. The non-parametric memory consists of efficiently stored documents that can be accessed to retrieve relevant knowledge. Conversely, parametric memory refers to the knowledge that is learned and stored within the parameters (weights) of a large language model during training.

The first step of RAG is to encode the natural language query. This encoded query representation is then used in the next module: the document selector [29]. The document index consists of the efficiently stored documents. The document selector selects the documents or parts of documents from the document index that are most relevant to the query. These are then augmented into the prompt as relevant context before being passed to the LLM.

The benefit of this approach is that parametric and non-parametric memory have their own strengths, making their combined ability even stronger. The non-parametric memory added by RAG helps combat LLM hallucinations, and is also easily replaceable with another document index without needing to retrain the LLM [29].

2.6 Coding assistant

Coding assistants are tools used to assist in coding tasks such as code generation and refactoring. AI-based code assistants have increased in popularity, acting as useful coding tools for increasing the efficiency of the user [12]. The coding assistant used in this work is added as an extension in the integrated development environment. Whereas RAG is commonly used for natural language, coding assistants tend to use RAG on code. As code also consists of text, it can be tokenized in a similar manner to what is described in Section 2.4, though based on

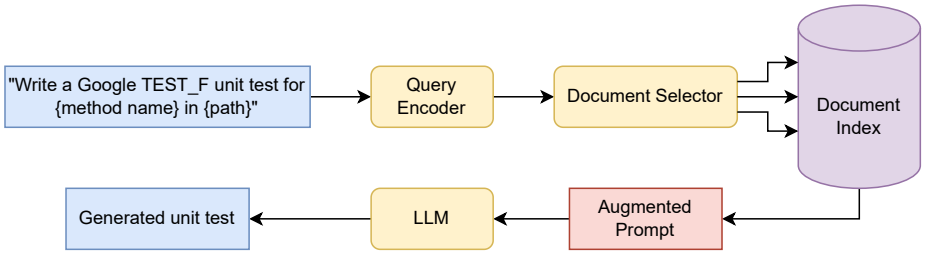


Figure 2.5: Overview of retrieval-augmented generation. Figure inspired by Lewis et al. [29]

meaningful units in code such as keywords, identifiers, operators and literals. These tokens can then be embedded in a vector space in the same way as natural language [2].

The document index described in Section 2.5, can be replaced by a code index, such as an entire code base. The document selector can then select the parts of the code base that it considers most relevant to the input string, adding them to the prompt, before letting the LLM generate an output. For code generation, coding assistants may use an LLM solely pre-trained on generating code.

2.7 Technology adoption frameworks

Numerous theoretical frameworks have been developed to understand how new technologies are adopted and accepted by users [1][4][15][40][49]. These frameworks aim to reveal the factors that drive or hinder the integration of these technologies into everyday practices. Among these frameworks, the *technology acceptance model* (TAM) [15], the *diffusion of innovation theory* (DoI) [40], and the *unified theory of acceptance and use of technology* (UTAUT) [49] provide insights into the benefits and challenges of generative AI for software testing.

2.7.1 Technology Acceptance Model

TAM is a well-established framework used to understand how users come to accept and use new technologies. It was developed by Davis [15] who built upon the theory of reasoned action [18], that states that an individual's behavior is determined by their intention to engage in that behavior. TAM adapts this foundation specifically for technology acceptance by proposing that user adoption is primarily influenced by users' perceptions rather than system characteristics alone.

At its core there are two main constructs: *perceived usefulness* and *perceived ease of use*. The former refers to the degree to which a person believes that using a particular system would enhance their job performance, and the latter refers to the degree to which a person believes that using the system would be free of effort [15]. According to TAM, the acceptance of technology occurs through a sequential three-stage process. First, external system characteristics activate cognitive processes in users (perceived ease of use and perceived usefulness). Second, these perceptions shape the users' intentions toward the technology. Finally, these attitudes determine actual usage behavior. This framework suggests that if generative AI tools for software testing are perceived as easy to use, they are more likely to be considered useful, thereby increasing the likelihood of acceptance and integration into users' software testing workflows.

2.7.2 Diffusion of Innovations

DoI is a theory developed by Rogers [40] that focuses more on the characteristics of the technologies themselves rather than the user perceptions, which TAM focuses on. There are five key attributes affecting the adoption of new technologies according to this theoretical model:

Relative advantage, *Compatibility*, *Complexity*, *Trialability* and *Observability*. Relative advantage refers to the perceived superiority of a new technology compared to what it replaces. Compatibility represents how well a new technology aligns with adopters' existing values, experiences, and needs. Complexity indicates the perceived difficulty in understanding and using a new technology. Trialability describes to which extent a new technology can be experimented with before full adoption. Observability reflects how visible the new technology's results are to others. Relative advantage, compatibility, trialability, and observability are positively correlated with adoption rates, meaning higher levels of these attributes lead to faster adoption. Complexity, on the other hand, is the only attribute negatively related to adoption, meaning more complex innovations are adopted more slowly.

These attributes are important when considering the adoption of generative AI tools in software testing. For example, the relative advantage of having AI generate tests instead of writing them manually can be a strong motivator for adoption. Compatibility with existing tools and processes, as well as the complexity of understanding and implementing AI solutions, are critical factors that can either help or hinder the spread of these innovations within the software testing domain.

2.7.3 Unified Theory of Acceptance and Use of Technology

UTAUT is a framework developed by Venkatesh et al. [49] that combines eight previous technology acceptance models into one unified theory. The UTAUT model has four key constructs that influence behavioral intention and usage behavior: *performance expectancy*, *effort expectancy*, *social influence* and *facilitating conditions*. Performance expectancy refers to the degree to which an individual believes that using the technology will enhance their job performance, similar to the perceived usefulness construct of TAM. Effort expectancy represents the perceived ease of using the technology, similar to the perceived ease of use construct of TAM. Social influence reflects how users perceive that important others believe they should use the new technology. In this thesis, the facilitating conditions construct is particularly emphasized as it provides a crucial organizational perspective that complements the more individually-focused TAM and the innovation-focused DoI frameworks. Facilitating conditions are defined as "the degree to which an individual believes that organizational and technical infrastructure exists to support the use of the system" [49]. Unlike the other three UTAUT constructs that primarily influence behavioral intention, facilitating conditions is the only construct that directly impacts actual usage behavior. This makes it especially relevant when examining the organizational context of generative AI adoption in software testing environments. For instance, even if software testers perceive AI tools as useful and easy to use, inadequate organizational support such as insufficient training, lack of clear guidelines or incompatibility with existing workflows, can pose significant hindrance to adoption.

2.7.4 Combining the frameworks

By combining TAM, DoI, and UTAUT in this thesis, we aim to create a comprehensive approach to understanding the benefits and challenges of generative AI adoption in software testing. The choice of these frameworks over others was deliberate and guided by their constructs that, in our belief, have the most potential to capture benefits and challenges. Each framework offers a different perspective. TAM looks at how useful and easy-to-use individuals find the technology. DoI focuses on the technology's own features and qualities. UTAUT, especially through its facilitating conditions concept, examines how organizational support affects adoption. Together, these frameworks allow us to analyze adoption factors from individual, technological, and organizational perspectives, with the aim of providing a more nuanced and complete understanding than a single theory could offer.

2.8 Related work

This section explores previous research in the fields of unit test generation (Section 2.8.1) and perceptions about generative AI (Section 2.8.2), and their relation to our work.

2.8.1 Unit test generation

Unit test generation has traditionally been a significant focus in software engineering research, with techniques evolving from symbolic execution and search-based optimization to the use of deep learning models. This section presents relevant related work and Table 2.1 summarizes the differences between the related work and our work. The table compares the work in terms of the content included in the prompt that is passed to the LLM, the architecture or method that is used, and the programming language the approaches investigated.

Early approaches

Early efforts utilized symbolic execution tools such as KLEE [35][50] and evolutionary algorithms like EvoSuite [19], aiming to maximize line coverage through systematic exploration or optimization-based approaches. While effective in terms of coverage, these methods often produced tests that were difficult for developers to maintain and interpret, limiting their utility as they would require additional resources in maintenance.

The rise of deep learning methods introduced a paradigm shift by framing unit test generation as a sequence-to-sequence learning problem. The AthenaTest [47] approach demonstrated promising results by generating syntactically and semantically correct tests in Java. However, this method requires fine-tuning a pre-trained model. Furthermore, AthenaTest is based on BERT, an encoder-only model, whereas the decoder-only architecture has shown more promising results and as described in Section 2.4, it has become the de facto model type for LLMs.

Decoder-only approaches

Other approaches explore the viability of generating unit tests without the need of fine-tuning. Bareiß et al. [5] evaluated Codex [10], a state of the art decoder-only model at the time, on three different Java tasks, one being test case generation. The paper is a systematic study, proposing a few-shot learning approach to code generation and comparing it to a traditionally built tool and found that the former outperforms the latter.

Few-shot learning was also used in MuTAP [14], an approach introduced by Dakhel et al., which uses mutation testing to improve the decoder-only LLM's ability to detect bugs in Python. As mutation testing creates mutants of the test, this approach aims to add the mutants to the prompt in order to more consistently reveal bugs.

Schäfer et al. [44] evaluated TestPilot on JavaScript, using an approach that does not provide the decoder-only LLM with context about the specific MUT, instead relying on an example MUT and usage examples extracted from documentation. Failed tests are regenerated by prompting the model again using the error message. The approach shows promising improvements in coverage, but the lack of both fine-tuning and any examples in the prompt limits the approach.

While our work also uses a decoder-only approach, it differs from the other approaches by using RAG to optimize the prompt content. Additionally, our investigation is based on C++ code, in contrast to Java, JavaScript, and Python, which the other work is based on.

Importance of features and high-quality database

Extracting relevant information is an important task, as there is no obvious order in what features are considered the most relevant. Yang et al. [51] explored the importance of different

Table 2.1: Comparison of related work on unit test generation

Study	Prompt content	Architecture	Language
EvoSuite [19]	-	Evolutionary algorithm	Java
AthenaTest [47]	MUT, MUT class	Fine-tuned encoder-only	Java
Bareiß et al. [5]	MUT, Example MUT-test pairs	Decoder-only	Java
MuTAP [14]	Instructions, Examples, Mutants	Decoder-only	Python
Schäfer et al. [44]	MUT, Usage examples	Decoder-only	JavaScript
Our work	MUT, RAG code-snippets	Decoder-only	C++

features and found that other methods in the same class as the MUT contribute more "to the syntactic validity of generated unit tests" than the other four studied code features (MUT parameters, class constructor, class fields and related classes). The same paper also found that the RAG approach used in the paper does not make any significant contribution to the LLMs ability in generating unit tests. The authors underscore the importance of having a retrieval database specifically customized for unit test generation with high-quality examples that resemble the ideal test that would be generated by the LLM.

2.8.2 Perceptions of generative AI

An overview of relevant studies exploring perceptions of generative AI are shown in Table 2.2. We categorize research about perceptions of generative AI broadly into two categories based on their *aim*: studies investigating the factors that influence technology adoption (AF in the table), and those exploring users' perceived benefits and challenges when using the technologies (BC in the table). The former use established theoretical frameworks to understand what motivates users to adopt generative AI tools, while the latter focus on the perceptions of users who have used generative AI in their work. The studies are further distinguished based on several characteristics: *focus area*, *data collection*, *data type*, *analysis method*, *sample size* and *research setting*. Their focus area highlights which aspect of software engineering each study targets, while the data collection describes the methodological strategies used to gather information from participants. The data type indicates whether researchers collected quantitative data such as ratings or multiple-choice answers, or qualitative data such as open-ended responses. The analysis method details the techniques used to process and interpret the data. Additionally, the sample size provides the number of participants in each study, and the research setting differentiates between studies conducted in academic settings and those involving professionals from industry. As shown in the table, our work differs from existing research primarily in its focus on software testing rather than general software development, offering unique insights through qualitative interviews with industry professionals in software testing.

Factors influencing the adoption of generative AI

Kim et al. [26] conducted a quantitative survey of 283 programmers in the software development industry, analyzing the data through statistical methods to extend TAM to include factors such as perceived enjoyment. They discovered that both perceived usefulness and enjoyment shape attitudes toward AI coding assistants, with enjoyment playing a surprisingly strong role.

Russo used a mixed-methods approach, combining an initial qualitative survey of 100 engineers with a quantitative validation survey of 184 more professionals from industry, to study generative AI adoption. Using thematic and statistical analysis methods, and grounding his work in theories like the TAM and DoI, Russo [42] found that perceived usefulness

Table 2.2: Comparison of related work on perceptions of generative AI

Study	Aim	Focus area	Data collection	Data dype	Analysis method	Sample size	Research setting
Kim et al. [26]	AF	SD	S	Quan	SA	283	Ind
Russo [42]	AF	SD	S (2x)	Qual, Quan	TA, SA	100 + 184	Ind
Chan and Hu [9]	BC	SD	S	Qual, Quan	TA, DA	399	Acad
Liang et al. [30]	BC	SD	S	Qual, Quan	OC, SA	410	Ind
Coutinho et al. [13]	BC	SD	S	Qual	TA	13	Ind
Hamza et al. [22]	BC	SD	W	Qual	TA	22	Ind
Klemmer et al. [27]	BC	SD	I	Qual	TA	27	Ind
Our work	BC	**ST**	I	Qual	TA	12	Ind

Aim: AF = Adoption Factors; BC = Benefits and Challenges
Focus Area: SD = Software Development; ST = Software Testing
Data Collection: I = Interviews; S = Surveys; W = Workshop
Data Type: Qual = Qualitative; Quan = Quantitative
Analysis Method: TA = Thematic Analysis; SA = Statistical Analysis; DA = Descriptive analysis; OC = Open Coding
Research Setting: Ind = Industry; Acad = Academia

and perceived ease of use, two key constructs of TAM, drive adoption less than the perceived compatibility with existing workflows.

These adoption studies provide a broader context for our work, but they lack a specific focus on software testing and on identifying the perceived benefits and challenges with the technology, which our research addresses directly through qualitative interviews with industry professionals.

Perceived benefits and challenges using generative AI

Several studies explore the perceived benefits and challenges of generative AI in software development, offering insights that, while not necessarily specific to software testing, are relevant to our study.

Chan and Hu [9] surveyed 399 university students to examine their views on generative AI, such as ChatGPT, in academia. Using both qualitative and quantitative data analyzed through thematic and descriptive analysis, they identified benefits including personalized learning support, writing and brainstorming support, research and analysis support, visual and audio support, and support for repetitive tasks. However, they also noted challenges such as accuracy and transparency concerns, privacy and ethical issues, over-reliance that impacts critical thinking and creativity, career prospect uncertainties, misalignment with human values, and uncertain policies around AI usage.

Liang et al. [30] surveyed 410 developers from industry about AI programming assistants like GitHub Copilot, using open coding [46] to analyze qualitative data and statistical analysis to analyze quantitative data. They highlighted benefits such as convenience and efficiency (speeding up task completion via autocompletion and reducing online searches), with successful use cases including generating repetitive code, simple logic, quality assurance, brainstorming proof-of-concepts, learning new languages, recalling syntax, improving efficiency, generating documentation, and ensuring code consistency. Challenges included quality and control issues, usability problems (unclear input-output relationships and unusable outputs), and struggles with understanding and modifying output due to unfamiliar APIs or complex structures, alongside privacy concerns about exposing proprietary code.

Coutinho et al. [13] conducted a study with 13 industry professionals, collecting qualitative data through a survey that was analyzed using a thematic analysis. They reported benefits like time optimization (automating routine tasks such as document generation), versatility (supporting ideation, code construction, and formal writing), and improved commu-

nication. Challenges included reliability and refinement issues (needing manual adjustments for complex tasks), difficulties in prompt crafting and source verification, and security concerns when handling sensitive data.

Hamza et al. [22] explored human-AI collaboration via a workshop with 22 industry engineers, using thematic analysis of qualitative data. They identified benefits such as human-AI collaboration enhancing problem-solving, and AI capabilities like code generation and assistance, documentation and explanation, programming language versatility, code review and optimization, and testing and quality assurance. Challenges included code security risks, inability to debug, struggles with complex logic, a learning curve for adoption, and social loafing from over-reliance.

Klemmer et al. [27] conducted 27 semi-structured interviews with industry professionals, using thematic analysis to explore AI in secure software development. They found benefits like increased productivity and efficiency (automating mundane tasks), versatile usage across tasks (such as coding, documentation, and vulnerability detection), and learning and skill enhancement. Challenges included security concerns (mistrust in AI-generated code due to vulnerabilities or misuse), privacy and data leakage risks, quality and reliability issues requiring manual review, organizational constraints, and potential job impact.

Our work is unique from these studies in that it places a greater emphasis on software testing, specifically focusing on identifying the perceived benefits and challenges within this domain through qualitative interviews with industry professionals.

3 Method

This chapter describes the methodology used to investigate the two research questions. In Section 3.1, we present the extraction process of the industrial dataset from the company's code base. Section 3.2 goes over the methodology used to utilize the commercial coding assistant for generating the unit tests to be evaluated in this thesis. In Section 3.3, the interview procedure to gather the qualitative data of the user study is explained. Finally in Section 3.4, the thematic analysis methodology is presented that is used to identify the perceived benefits and challenges as framed in RQ2.

3.1 Data collection for unit test generation

To generate unit tests and compare them with gold standard tests, a dataset consisting of pairs of an MUT and a matching unit test is necessary. To collect such a dataset, we use the *tree-sitter-cpp* library[2] to create an abstract syntax tree of a C++ repository. The library creates a tree with nodes representing files and functions. We use this to extract the functions or methods from the files as described by Algorithm 2.

- The first step of the algorithm is to enter every file that contains software under test and extract all methods under test.
- The next step is to similarly extract unit tests from the test file, with the added requirement that the extracted unit test must start with the macro TEST_F.
- Once all MUTs and unit tests have been collected, we match them in pairs. An MUT is matched to a unit test if the unit test calls the MUT.
- In the final step, we remove entries that we suspect would have too long sequence length for the LLM, based on the LLM's context length.

This filtering leaves us with a final dataset of approximately 12,000 data points.

[2]https://pypi.org/project/tree-sitter-cpp/

Algorithm 2: Data collection

```
Data: Code files and Test files
Result: Matches between methods under test and unit tests
method_under_test_list ← Empty Set;
foreach code_file do
    foreach method_under_test in code_file do
        method_under_test_list.add(method_under_test);
    end
end
test_list ← Empty Set;
foreach test_file do
    foreach unit_test in test_file do
        if unit_test starts with "TEST_F" then
            test_list.add(unit_test);
        end
    end
end
matches ← Empty Set;
foreach unit_test in test_list do
    foreach method_under_test in method_under_test_list do
        if test calls method_under_test then
            matches.add((method_under_test, unit_test));
        end
    end
end
remove_too_long_entries();
```

3.2 Unit test generation

This section provides a description of the methodology used to prepare the dataset, perform the unit test generation, and evaluate the generated unit tests.

Avoiding data leakage The coding assistant uses indexing of the entire repository as described in Section 2.6. This means that the gold standard tests would already be within the coding assistant context when generating new tests if we start indexing directly. This issue could be resolved by removing all the gold standard tests from the repository before letting the coding assistant index the repository. However, this would lead to another problem: it would disallow the coding assistant to learn the structure of tests used at the company, and would not reflect a real-world use case where there tend to be existing unit tests in the repository already.

To avoid both of these problems, we split the dataset into two halves: the *generation half* and the *context half*. We remove the gold standard unit test for the data points in the *generation half*, while keeping the gold standard unit tests data points in the *context half* for the indexing to learn from. After generating unit tests for the *generation half*, we repeat the experiment with reversed roles for the two dataset halves.

Generating unit tests The version of the coding assistant used at the company does not have a way to generate multiple unit tests through an API. Instead, the only ways to generate unit tests are clicking on buttons or using keyboard shortcuts. We automate the unit test generation using a script that simulates such keyboard shortcuts. The script automatically prompts the model using the prompt "Write a Google TEST_F unit test for {method_name} in {path}".

Once the generation is complete, we post-process each output by extracting a single test from it, since each output string may contain multiple generated unit tests. We do this by finding the start and end of the unit test. We identify the start by finding the *TEST_F* macro described in Section 2.2 and the end by finding where all the opened curly braces have been closed. Additionally, we remove all comments from the generated unit tests.

Evaluating generated unit tests To evaluate the generated unit tests, we attempt to compile them. From the generated log, we extract whether the test compiled.

We calculate the similarity between the generated tests and their gold standard equivalents using the three similarity metrics described in Section 2.3, that is cosine similarity, Jaccard similarity and Levenshtein similarity. As explained in Section 2.3, these similarity metrics are different ways to describe how closely the generated tests resemble their gold standard equivalents. This will help determine the quality of the generated tests, since the company has high quality requirements for their existing tests. In order to interpret the similarity scores in terms of quality, we randomly sample data points corresponding to 3% of the total dataset. For each generated test in the sample, we rate its resemblance to its gold standard equivalent, rating the resemblance as either low (1), medium (2), or high (3).

Thresholds For each similarity metric, we determine two thresholds by maximizing the macro F1-score for our resemblance categories (low, medium, high). The *low-medium* threshold (T1) distinguishes low resemblance cases from medium and high resemblance cases, while the *medium-high* threshold (T2) separates low and medium resemblance cases from high resemblance cases. We employ a joint optimization approach using grid search over all possible threshold pairs (T1, T2) with the constraint T1 < T2. At each point in the grid, we evaluate the three-class macro F1-score, which averages the F1-scores across all three resemblance categories (low, medium, high). The threshold pair that yields the highest macro F1-score is selected as optimal.

3.3 Interviews

The user study for this thesis involved interviewing software engineers who have experience with software testing and have used generative AI. To maximize participant recruitment, it was not required that they use generative AI specifically for software testing tasks. However, they needed to have experience with both software testing and generative AI independently. With these recruitment constraints, twelve participants volunteered to be interviewed. According to prior research [20], data saturation, when further interviews no longer reveal new themes or information, is usually achieved after approximately twelve interviews. Thus, a sample of twelve participants in this study is to be considered adequate to provide qualitative insights. The interviews were designed using the theoretical frameworks of the TAM, DoI and UTAUT, using a similar approach as Russo [42]. However, unlike Russo's survey approach, we opted for interviews to delve deeper into the perceived benefits and challenges associated with generative AI tools.

The interviews were structured but allowed participants to elaborate on their responses. We also posed follow-up questions related to topics that emerged during the interviews that were of interest for the thesis. An interview guide was developed and adhered to by the two authors of this thesis, who conducted the interviews. The initial set of questions was refined in collaboration with our supervisor at the company to align with the company's objectives.

Interview design Initially, the interview questions were designed to address all major constructs and attributes of TAM and DoI. However, after several revisions, the questions were tailored more towards identifying the benefits and challenges of generative AI tools, rather than strictly adhering to the frameworks. This involved the exclusion of some questions in

favor of questions related to UTAUT as well as questions that were deemed more relevant for the company.

Interview structure We began each interview by asking the participants how many years of experience they have in their professional career, in roles involving software testing and in using generative AI. The questions and structure of the remaining interviews are presented in Figure 3.1. In order to maximize participant engagement, a branching interview structure was designed, that tailored questions according to the participants' use of generative AI. Specifically, if participants indicated that they did not utilize generative AI in their software testing tasks, the interview questions were slightly adjusted to explore their reasons for not using it and to discuss potential tasks for which they could consider utilizing generative AI. All interviews were conducted via online meetings. Participants were informed about the topic, procedure, anonymity, and their right to withdraw their participation at any time. Informed consent was obtained both prior to and at the start of the interviews, and each interview was recorded.

The interview questions used in the study and their relationship to relevant theoretical frameworks, where applicable, are listed in Table 3.1. Only the questions from the primary (left) path of the interview guide are listed, alternative variants of the questions can similarly relate to the frameworks.

Table 3.1: The interview questions used and what they aim to reveal along with references to the theoretical frameworks described in Section 2.7

Question	Aim
1. Do you use generative AI for any software testing tasks?	This question helps determine the direction of the interview, guiding the interviewer on which path to follow.
2. For what software testing tasks do you use generative AI?	This question identifies specific tasks where generative AI is applied, offering insights into its perceived benefits. The use of generative AI in these tasks suggests advantages, aligning with the concept of relative advantage in the DoI framework. Additionally, this usage reflects perceived usefulness in the TAM, indicating enhanced performance. Follow-up questions about tasks where generative AI is not used can reveal perceived challenges.
3. To what extent do you think using generative AI increases your efficiency in software testing tasks?	Adapted from Russo's work [42], this question measures how generative AI affects the productivity of software testers, relating to perceived usefulness in TAM. An affirmative response suggests a perceived benefit, while a negative response indicates a perceived challenge.
4. What concerns do you have about using generative AI in software testing tasks?	This question, modified from Russo's work, identifies perceived challenges associated with using generative AI. It relates to the construct of complexity in the DoI framework.
5. What potential benefits do you think generative AI offers over other more traditional methods in software testing?	This question, also modified from Russo's work, seeks to identify perceived benefits of using generative AI, relating to the construct of relative advantage in the DoI framework.
6. To what extent do you feel your organization is supportive of adopting generative AI as a standard software testing technology?	Modified from Russo's work, this question explores how organizational support impacts the adoption of generative AI tools among software testers, based on the construct of facilitating conditions in the UTAUT framework.
7. What resources or support do you feel are necessary to effectively use generative AI in your software testing tasks?	This follow-up question addresses both company needs in identifying necessary resources or support and is based on the construct of facilitating conditions in the UTAUT framework.
8. Can you share any specific failures you've experienced by using generative AI in your software testing work?	This question aims to identify perceived challenges of using generative AI for software testing, based on the construct of observability in the DoI framework.
9. Can you share any specific successes you've experienced by using generative AI in your software testing tasks?	This question seeks to identify perceived benefits of using generative AI for software testing, also based on the construct of observability in the DoI framework.
10. What is your opinion on using generative AI for unit test generation?	This question connects RQ2 with RQ1, allowing a comparison between software testers' perceptions of unit test generation and the quality of unit tests generated as explored in RQ1.

3.4 Thematic analysis

After completing each interview, the interviewer transcribed the recording. Since there were two interviewers involved in conducting the interviews, the one who did not conduct and transcribe a particular interview reviewed the transcription to ensure its accuracy. After each interview had been transcribed and the transcriptions had been reviewed, a thematic analysis

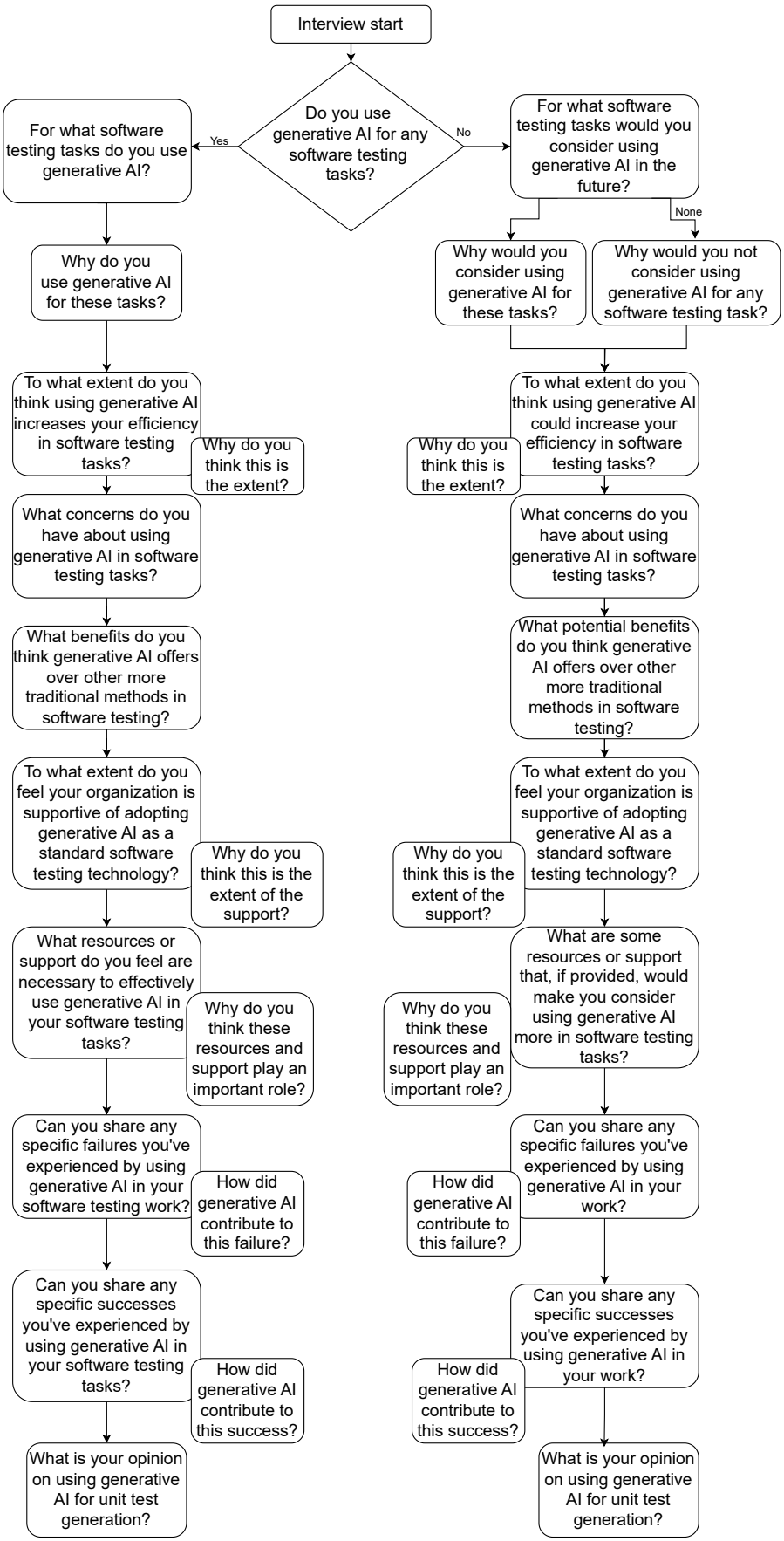


Figure 3.1: Interview questions and structure

was conducted on the qualitative data. This method involves identifying patterns within the data, analyzing these patterns and finally reporting on the patterns, called themes. Our analytic interest was guided by RQ2 in the sense that we aimed to identify and analyze the perceived benefits and challenges, rather than seeking to reveal all topics that were present in the dataset. With this in mind, a theoretical thematic analysis was conducted as described by Braun and Clarke [8], using methodology that is divided into six phases presented below. This variation of thematic analysis is a deductive approach that is driven by our interest in RQ2. Our usage of the six phases that form a thematic analysis is described below.

Phase 1: Familiarization with the data

Conducting the interviews followed by the subsequent transcription and proof-reading of transcriptions meant both authors gathered a comprehensible familiarization with the data. Attention was paid specifically to content relating to the perceived benefits and challenges of generative AI tools in software testing. Some early notes were made of some initial patterns that could be relevant for our analysis.

Phase 2: Generating initial codes

Coding involves extracting excerpts from the data (transcriptions) and assigning one or more *codes* to them. A code is a brief description that highlights an important feature of the excerpt relevant to the subsequent analysis. The purpose of this phase is to organize the data into groups that facilitate the next stage of analysis.

The segmentation of data was aimed to achieve excerpts that captured "the most basic segment, or element, of the raw data or information that can be assessed in a meaningful way regarding the phenomenon", as described by Boyatzis [7]. This process was further guided by our RQ2, which aims to reveal perceived benefits and challenges. Thus the approach was guided by the goal of extracting concise and standalone excerpts that clearly conveyed a single perceived benefit or challenge related to generative AI in software testing. To achieve this, we adopted a granularity aiming to allow each excerpt to be understood independently, minimizing the need for additional context. This involved selecting excerpts that were sufficiently detailed to capture the essence of participants' perceptions without introducing ambiguity by touching upon multiple aspects or meanings.

The classification of excerpts into perceived benefits or challenges was based on the content but also within the context the excerpts were expressed. Excerpts were categorized as perceived benefits if they highlighted advantages, improvements, or positive outcomes associated with generative AI tools. Conversely, excerpts were classified as perceived challenges if they pointed to difficulties, limitations, or negative impacts experienced or anticipated by participants. To ensure consistency and reliability in the classification process, each excerpt was reviewed and validated. In a few cases, if an excerpt required additional context from the interview to be fully understood or to clarify the code, this context was added as a side note to the excerpt.

Each excerpt was often assigned multiple codes when distinct features were present. Once all excerpts were coded by one author, the other author validated each code and suggested changes where there was disagreement or room for improvement. The outcome of this phase was a dataset in which each data point consisted of an excerpt, the participant who expressed it, any additional context needed, one or multiple codes assigned to that excerpt and the classification of each excerpt being a perceived benefit or a perceived challenge. Table 3.2 illustrates three excerpts from our dataset with their assigned codes (second column) after the second phase, for demonstration purposes.

Table 3.2: Three excerpts in our dataset that have been assigned codes and themes

Excerpt	Code(s)	Theme
I think for example one big problem is they have some token limitation.	token limitation	Practical Integration Barriers
...the ability to have people that are not experts on programming to actually develop.	empowering through AI, democratization of programming, accessibility	AI Empowerment
To be honest not so much. I tried to use it before, and most times it works well and sometimes not. It like stopped or something.	mixed reliability	Accuracy and Reliability of AI Output

Phase 3: Searching for themes

This phase involved grouping and combining codes into broader, overarching themes. The data points were grouped based on their classification of being a perceived benefit or perceived challenge. This initial separation of codes was useful to help form distinct themes related to perceived benefits or challenges. Thereafter, data points with the same codes were grouped together, followed by combining groups of similar codes to form broader set of codes. Our running example of three data points that have been assigned themes following the completion of the third phase is shown in Table 3.2.

Phase 4: Reviewing themes

All data points within each theme were revisited and the codes assigned to the data points were evaluated as to whether a coherent pattern among them was present. In some cases when a data point was deemed not to fit within the theme, the data point was moved to a more suitable theme or was part of the creation of a new theme entirely.

Phase 5: Defining and naming themes

Having a set of themes that have been reviewed and iterated on in the previous phase, Phase 5 is about naming them with the aim of capturing the essence of each theme, as described by Braun and Clarke [8]. At this stage, an iterative process involving repeated cycles through Phase 3 to Phase 5 was conducted multiple times, ensuring that the theme names effectively conveyed the shared meanings of the excerpts they contained.

Phase 6: Producing the report

Braun and Clarke [8] present the sixth and final phase of the thematic analysis as enveloping both the final analysis and writing the complex story captured in the data, ensuring its validity. From each theme, we identified several benefits or challenges, and presented these by including supportive excerpts.

4 Results

This chapter presents the results of the performance evaluation as well as a thematic analysis of the interviews to answer RQ1 and RQ2, respectively.

4.1 Quality of generated unit tests

To evaluate the quality of the generated unit tests, this section presents measured results for the four evaluation metrics: compilation success rate, cosine similarity, Jaccard similarity and Levenshtein similarity.

4.1.1 Evaluation observations

After conducting the evaluation as described in Section 3.2, we classify each generated unit test into one of three categories:

- **Category I**: The test compiles and does not have any failing assertion.
- **Category II**: The test compiles, but has a least one failing assertion.
- **Category III**: The test does not compile.

The flow of test results is visualized in Figure 4.1. While almost two-thirds of the generated tests (63.91%) fail to compile, 36.09% of the generated tests compile successfully. As mentioned in Section 2.2, unit tests include assertions that fail if they receive unexpected values. Although some of these compilable tests (11.45% of total tests) have failing assertions, a majority (24.64% of total tests) do not fail any assertions, hence they pass successfully.

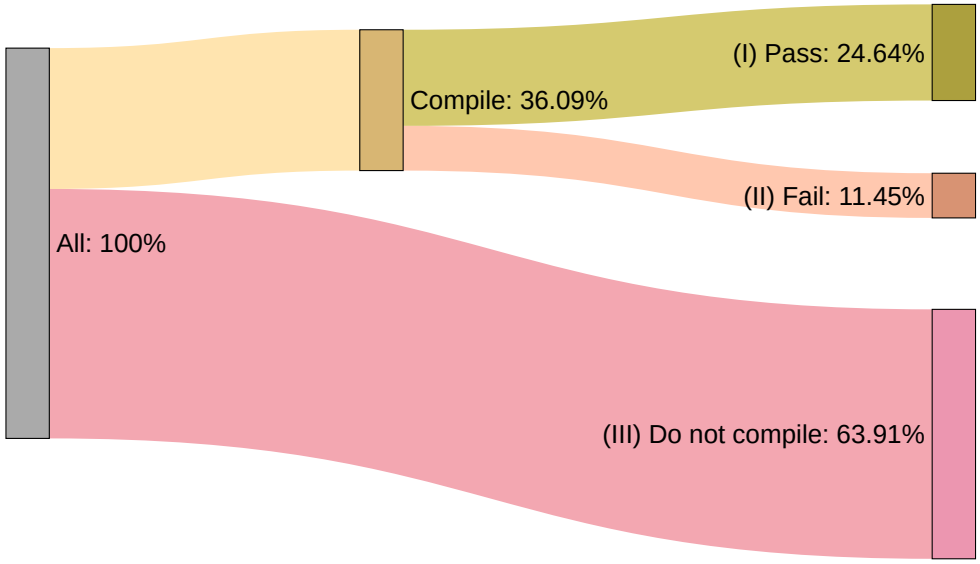


Figure 4.1: Flow of test results with regard to compilability and passability

Table 4.1: Frequency of the most common compilation errors for tests that did not compile

Error Message	**Frequency**
no matching function for call to {function}	21.11%
others	14.97%
{identifier} was not declared in this scope	13.36%
{class} has no member named {member}	12.99%
no match for operator {operator}	12.06%
{member} is not a member of {class}	8.25%
{function} is protected within this context	4.82%
{identifier} not been declared	3.48%
{identifier} is private within this context	2.00%
cannot convert {type1} to {type2}	1.64%

4.1.2 Errors

Recognizing that the major obstacle in generating unit tests appears to be lack of compilability, we find it important to explore the underlying reasons for compilation errors. Table 4.1 shows the frequency of the most common compilation errors for such tests. From the table, it becomes clear that the main issue is that the generated unit tests tend to include function calls, identifiers, classes and members that are not recognized in the current scope. Apart from these, access modifiers and unrecognized operators also seem to be common causes of compilation errors.

4.1.3 Similarity

As described in Section 2.3, each generated unit test is compared to its gold standard equivalent, in terms of cosine similarity, Jaccard similarity and Levenshtein similarity. In Table 4.2, statistics of similarity scores are presented for tests belonging to each test category. The calculated thresholds and the distribution of manually labeled resemblance ratings for each similarity metric are presented in Figure 4.2. This section discusses the results of all three similarity metrics.

cosine similarity The thresholds for this metric were 0.55 and 0.89 for the low-medium and the medium-high thresholds respectively, yielding a macro F1-score of 0.756. The mean and median cosine similarity scores between all generated tests and their gold standard equivalents are 0.49 and 0.45 respectively. This is slightly below the low-medium threshold of 0.55, indicating that the generated tests have low resemblance with the gold standard tests in terms of the usage frequency of key terms. However, looking at only tests in Category I, the mean similarity is 0.77, meaning there is medium resemblance to the gold standard tests. The median is even higher (0.90), meaning that at least half of the generated tests in Category I have high resemblance with their gold standard equivalents. This means that at least half of the generated tests in Category I have high resemblance with their gold standard equivalents. These medium to high cosine similarity scores of Category I indicate that tests that do compile and pass also tend to use similar keywords with similar frequencies as their gold standard equivalents.

Jaccard similarity This similarity metric established optimal classification thresholds at 0.26 for low-medium distinction and 0.60 for medium-high distinction, achieving an overall macro F1-score of 0.764. The mean Jaccard similarity is 0.35, falling in the medium resemblance range, and the median is 0.25, falling in the low resemblance value range. This suggests mixed performance overall, with medium keyword overlap on average but poor overlap for a majority of cases. Category I tests show significantly better performance at 0.64 and 0.67, respectively, both indicating high resemblance. The Jaccard similarity medians for Category II and Category III tests are remarkably low at 0.22 and 0.20, suggesting that tests that struggle with compiling and passing are generally those that have minimal keyword overlap with their gold standard equivalents, while successful tests demonstrate strong keyword similarity.

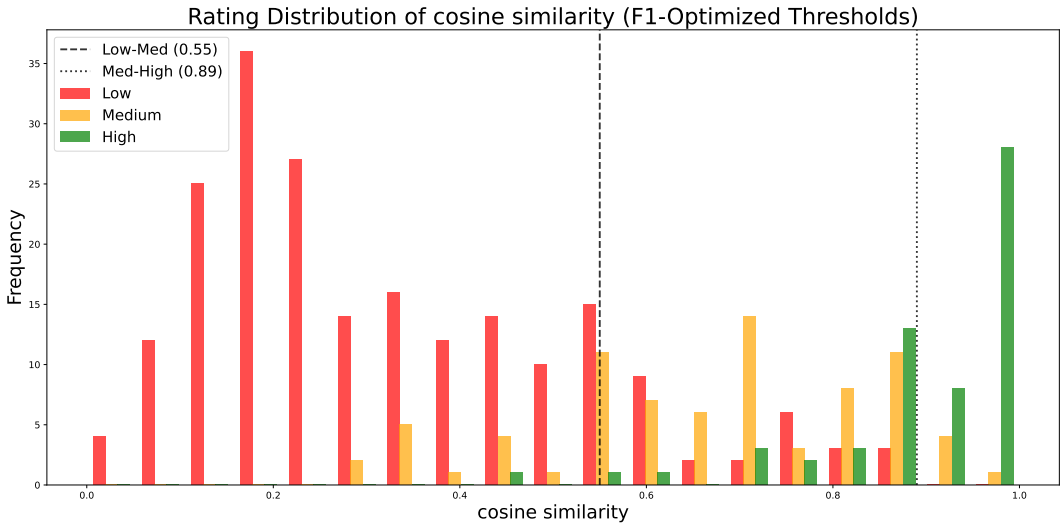
Levenshtein similarity This similarity metric had thresholds of 0.37 and 0.73 for the low-medium and the medium-high thresholds respectively, resulting in the highest combined F1-score of 0.785. The Levenshtein mean (0.43) falls in the medium resemblance range, and the median (0.36) falls in the low resemblance value range, indicating medium average similarity but poor character-level resemblance for most generated tests. For Category I tests, the mean (0.72) falls in the medium resemblance range, while the median (0.79) falls in the high resemblance range. This indicates that generated tests that do compile and pass require relatively few character-level operations to transform them into their gold standard equivalents, with at least half showing high character-level similarity.

4.1.4 Test length

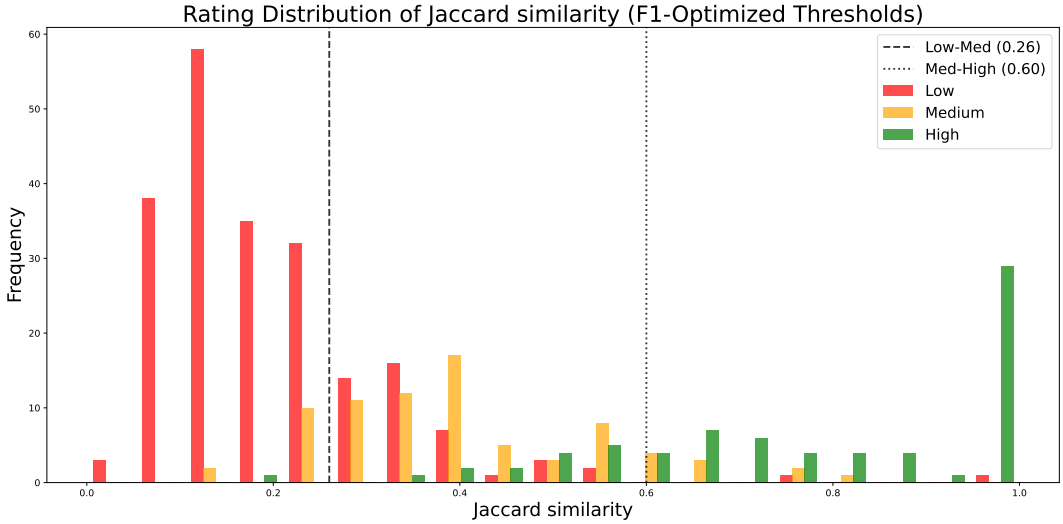
Intuitively, generated unit tests consisting of fewer lines of code should be less complex and thus of higher quality, although with some exceptions such as tricky corner cases. To validate this, we plot the distribution of test length in number of lines for the three categories as visualized in Figure 4.3. The plot shows that there is a general trend of shorter tests belonging to a more desired category. Similarly, Figure 4.4 shows that tests generated for shorter methods under test are more likely to be in Category I.

Table 4.3 shows statistics for each category in terms of number of lines in the generated test. Interestingly, tests from Category I have a max length of 61, meaning that no generated tests with more than 61 lines compile and pass.

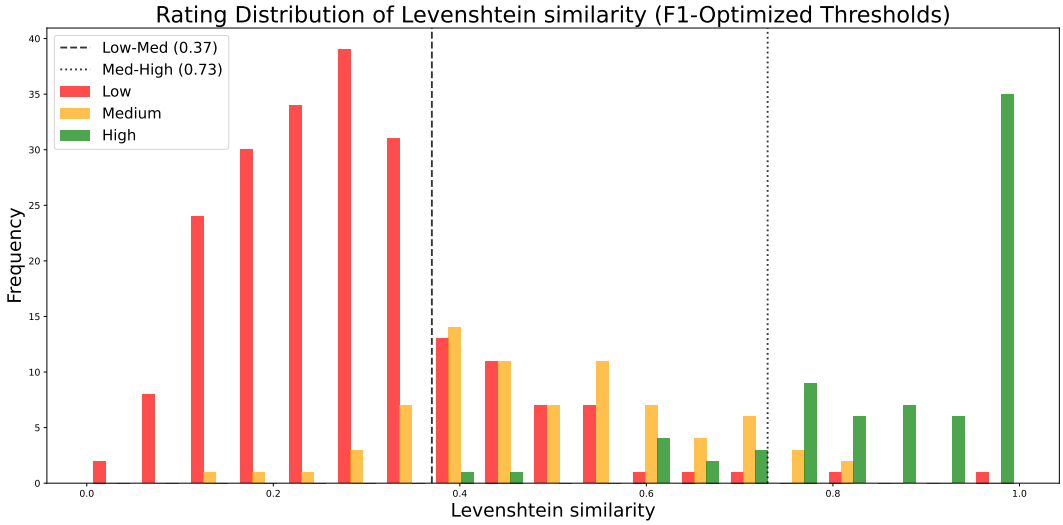
As we identify poor compilability in the generated tests, we investigate the correlation between test length and compilability. Figure 4.5 visualizes the percentages of compiled tests at each test length. The figure shows that single and double line generated tests almost always compile, before gradually dropping to an approximate 15%-20% compilation success rate at between 17 and 35 lines, and shortly thereafter dropping to zero compilation success rate for even longer tests except for a few outliers.



(a) Rating distribution of cosine similarity



(b) Rating distribution of Jaccard similarity



(c) Rating distribution of Levenshtein similarity

Figure 4.2: Rating distribution of the three similarity metrics

Table 4.2: Mean, median, and standard deviation of similarity scores for the three test categories in Section 4.1.1, classified as low (yellow), medium (light green), or high (green) resemblance based on metric-specific thresholds

		Category			
Similarity metric		**I**	**II**	**III**	**All**
cosine	mean	0.77	0.43	0.40	0.49
	median	0.90	0.37	0.34	0.45
	std dev	0.28	0.29	0.24	0.30
Jaccard	mean	0.64	0.30	0.25	0.35
	median	0.67	0.22	0.20	0.25
	std dev	0.31	0.22	0.17	0.27
Levenshtein	mean	0.72	0.38	0.33	0.43
	median	0.79	0.33	0.30	0.36
	std dev	0.27	0.22	0.19	0.27

Resemblance:

cosine: Low < 0.55, Medium $0.55 - 0.89$, High > 0.89

Jaccard: Low < 0.26, Medium $0.26 - 0.60$, High > 0.60

Levenshtein: Low < 0.37, Medium $0.37 - 0.73$, High > 0.73

Table 4.3: Statistics about the number of lines in the generated tests, for the three test categories described in Section 4.1.1

Statistic (number of lines)	**I**	**II**	**III**
mean	7.1	12.8	15.4
median	6	11	13
std dev	5.2	8.0	11.7
min	1	1	1
max	61	117	222

Lastly, an analysis of the difference in test length between generated tests and their gold standard equivalents reveals that out of all pairs of generated unit tests and equivalent gold standard tests, 74% differ in length with at most 3 lines. This indicates that the coding assistant is able to generate tests close to the company's standard test length.

Answer to RQ1

In terms of compilation success rate, we find that out of approximately 12,000 generated unit tests, approximately one-third compile, and one-fourth compile and pass. When comparing the generated tests with their gold standard equivalents using cosine similarity, Jaccard similarity, and Levenshtein similarity, we find that the generated tests on average have low to medium resemblance to their gold standard equivalents. However, for generated tests that compile and pass successfully, the resemblance is much higher.

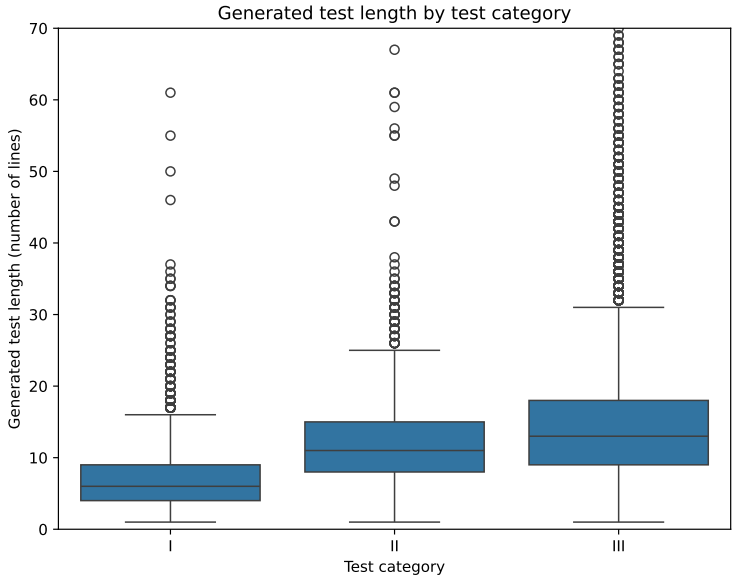


Figure 4.3: Boxplot illustrating the distribution of the generated tests' lengths for each test category

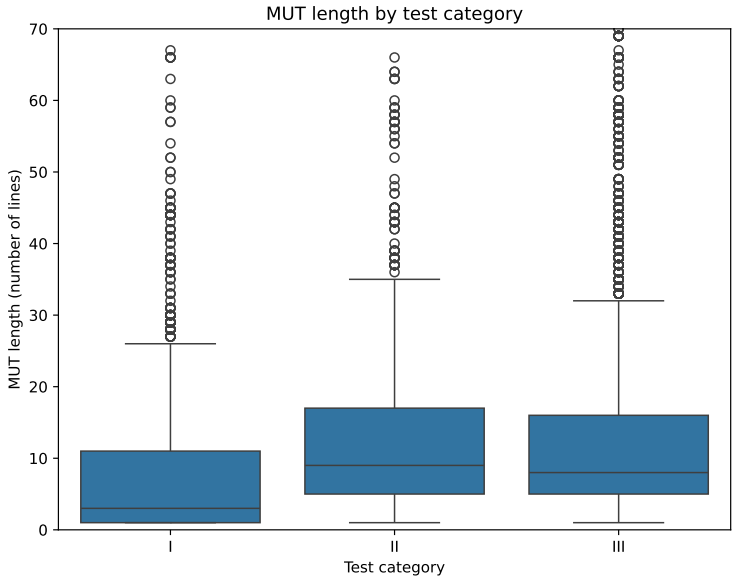


Figure 4.4: Boxplot illustrating the distribution of the MUT's lengths for each test category

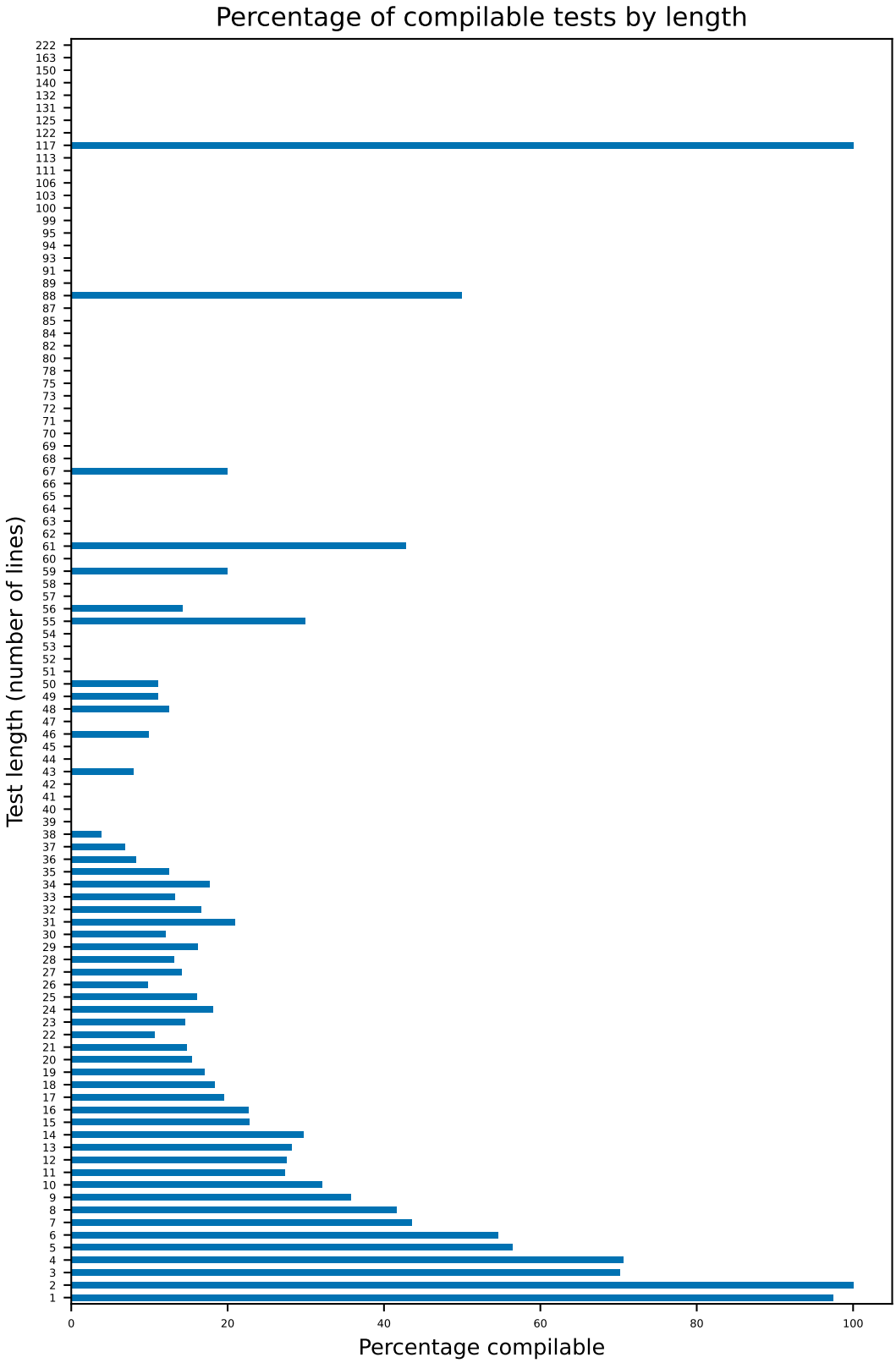


Figure 4.5: Percentage of compilable tests of different lengths

4.2 User study

In this section, we present the findings from our user study, focusing on the perceived benefits and challenges of generative AI tools in software testing. The results are organized to first provide background information on the participants, then an overview of the collected data, before finally providing an analysis of the themes that emerged from the data.

4.2.1 Participants

We interviewed a total of 12 software engineers, with varying levels of experience in their professional careers, software testing roles, and in their usage of generative AI tools.

- Professional career experience: Participants had an average of 15 years of professional experience, ranging from 3 to 24 years.
- Software testing experience: Participants reported a diverse range of experience in software testing, from those who have not held official software testing roles but have occasionally performed testing tasks as part of their development duties or in collaboration with QA engineers, to individuals who have worked in dedicated software testing roles for up to 24 years.
- Generative AI experience: Participants reported an average of 2 years of experience with generative AI, ranging from individuals who have only briefly experimented with these technologies to those who have been actively working with generative AI since the introduction of the transformer architecture in 2017 [48].

4.2.2 Data overview and processing

In total, the transcriptions of the interviews consisted of more than 18,000 words. After manually reading through the transcriptions and identifying excerpts that deemed to be hinting at a perceived benefit or perceived challenge, a total of 175 non-overlapping excerpts were identified to be used in the thematic analysis. Out of these excerpts, 81 (46%) were classified as perceived benefits, and 94 (54%) were classified as perceived challenges.

Codes and themes The visual representation of the resulting grouping procedure, which involved organizing excerpts based on similar codes as outlined in Section 3.4, is depicted in Figure 4.6. In this figure, light blue circles denote unique codes assigned to excerpts extracted from the transcriptions, while squares represent distinct groups. A circle (code) is connected to a square (group) via an edge if an excerpt within that group contained the particular code. Consequently, if excerpts from different groups share the same code, the code will have edges linking it to each relevant group. For instance, the circle connecting square 4 with square 6 indicates that at least two different excerpts were assigned this code, and that these excerpts have been categorized into different groups. This occurs because excerpts can encapsulate multiple meanings and thus be assigned multiple codes.

Our methodology aimed to achieve a level of granularity where most excerpts could be understood independently of additional interview context, to preserve the integrity of the data. Each excerpt was designed to capture a distinct essence of thought, classified as indicating either a perceived benefit or challenge. The codes assigned were intended to facilitate the separation of excerpts into distinct, non-overlapping groups, minimizing thematic overlap. In some cases, further segmenting the excerpts would have removed important context in which they were said, resulting in some excerpts being assigned codes that were less similar to each other and more likely to belong to different groups. As illustrated in Figure 4.6, we believe that the separation between groups is sufficient to form distinct themes, with most groups sharing only one or a few codes. A notable exception is group 3, which shares seven

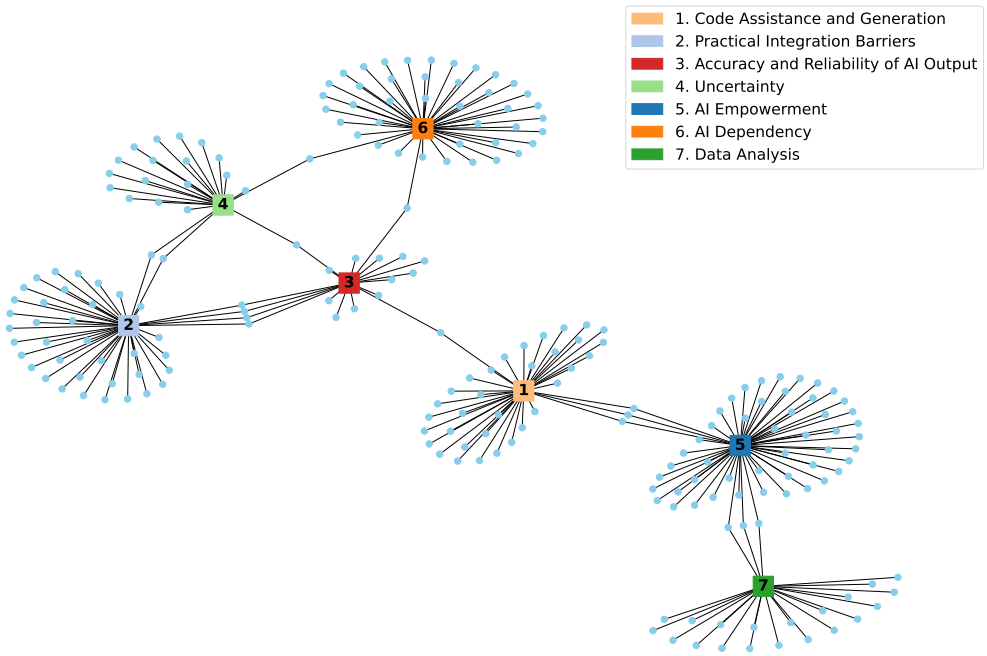


Figure 4.6: Codes (light blue circles) mapped to groups (squares)

Table 4.4: Theme names assigned to the groups, with each theme categorized as beneficial (B) or challenging (C)

Beneficial Themes	**Challenging Themes**
B: AI Empowerment	C: Accuracy and Reliability of AI output
B: Code Assistance and Generation	C: AI Dependency
B: Data Analysis	C: Practical Integration Barriers
	C: Uncertainty

codes with other groups, indicating that excerpts assigned these codes were more challenging to differentiate into groups.

The grouping process resulted in seven distinct themes, which were subsequently categorized as either *Beneficial Themes* or *Challenging Themes*, depending on whether the excerpts they contained were classified as perceived benefits or perceived challenges. Each theme was named through a review of the associated excerpts and their corresponding codes, with the goal of capturing the core idea of each theme and supporting the analysis of the data. The identified themes are presented in Table 4.4.

Participant contribution An overview of which participants contributed to each theme is presented in Table 4.5. An *X* indicates at least one excerpt of the participant was assigned to the particular theme. We note the uneven contribution among participants, which ranges from two themes (P3) to some participants that contributed to all themes (P4 and P8). Every participant also contributed to at least one theme categorized as either beneficial or challenging, showing all participants acknowledged at least some positive or some negative views.

Theme distribution The distribution of the number of excerpts across the themes is presented in Table 4.6. The theme "AI Empowerment" was discussed by the largest number of participants, while the theme "Data Analysis" was mentioned by only six participants. The table also displays the number of excerpts assigned to each theme. We observe an uneven distribution of excerpts, with the theme "AI Empowerment" containing almost three times as

many excerpts as the theme "Data Analysis." This could be a direct result of our grouping process, which is inherently interpretative and was not designed to create evenly distributed themes. Instead, our approach prioritized identifying meaningful patterns and relationships in the data, allowing themes to emerge themselves, regardless of how frequently each topic was discussed.

Table 4.5: Contributions of participants (P1–P12) to different themes, with each theme categorized as beneficial (B) or challenging (C)

Theme	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
B: AI Empowerment	X	X	X	X	X	X	X	X	X	X	X	
B: Code Assistance and Generation	X		X	X	X	X		X			X	X
B: Data Analysis				X				X	X	X	X	X
C: Accuracy and Reliability of AI Output	X	X		X	X			X		X		X
C: AI Dependency			X	X	X	X	X	X		X	X	
C: Practical Integration Barriers	X			X		X	X	X	X	X	X	
C: Uncertainty	X	X		X	X	X	X	X	X		X	X

Table 4.6: Themes categorized as beneficial (B) or challenging (C) ordered by the number of participants (P) and the number of excerpts (E)

Theme	# of P	# of E
B: AI Empowerment	11	37
C: Uncertainty	10	22
C: AI Dependency	8	30
C: Practical Integration Barriers	8	29
B: Code Assistance and Generation	8	28
C: Accuracy and Reliability of AI Output	7	13
B: Data Analysis	6	16

4.3 Thematic analysis

This section presents the thematic analysis conducted on the interview data. An overview of the beneficial themes and the benefits they reveal is shown in Table 4.7. Similarly, the challenging themes and the challenges they reveal is shown in Table 4.8. Each theme identified during the analysis is discussed, along with the benefits and challenges associated with the theme. These benefits and challenges are supported by excerpts extracted from the transcribed interviews. Some of the excerpts have been translated to English from Swedish. At the end of the section, we summarize the most significant benefits and challenges in order to address RQ2.

Table 4.7: Beneficial themes and the benefits perceived by engineers

Theme:	AI Empowerment	Code assistance and generation	Data Analysis
Benefits:	Time savings and efficiency improvement Organizational support and encouragement Democratization and accessibility AI collaboration Handling complexity and edge cases Focus on higher-level logic	Test generation Boilerplate and initial setup Code explanation and suggestions Code improvement and refactoring	Search and information retrieval Log analysis and troubleshooting

4.3.1 Theme: AI Empowerment

The thematic analysis revealed a significant theme around how generative AI tools empower software testers in their work, in different ways.

Benefit: Time savings and efficiency improvement

Participants reported large time savings when using generative AI tools for testing tasks. One participant directly stated that "...it saves a lot of time instead of I try to google some-

Table 4.8: Challenging themes and challenges perceived by engineers

Theme:	**Uncertainty**	**AI Dependency**
Challenges:	Security and data confidentiality concerns Lack of organizational guidance Lack of AI knowledge Transparency	Validation requirements Skill degradation Increased technical debt and maintainability issues Misapplication of AI in testing processes Potential change in review dynamics
Theme:	**Practical integration barriers**	**Accuracy and reliability of AI output**
Challenges:	Lack of support and knowledge sharing Organizational and resource constraints Token limitation and prompt engineering Diverse user needs	Output quality and reliability Complexity in software testing Domain knowledge limitations

thing or check Reddit or CUDA or anything of this stuff" highlighting how AI can replace time-consuming information searches. This sentiment was shared by another participant who estimated that AI could "...reduce the time by 50%. If I had to guess." The time savings reported were often of rather large magnitude. One engineer shared that "things that maybe would take me 2-3 days to figure out... can be done in like 2-3 hours, right? So it's a massive improvement in efficiency." Similarly, another participant noted that tasks that would normally take "30 minutes to do manually, with the help of generative AI, I can wrap up the understanding of logs in 10 minutes."

Benefit: Organizational support and encouragement

Participants consistently reported strong organizational support for AI adoption. One participant stated "there is very good support" while another noted "they are pushing quite a lot now" regarding organizational AI initiatives. One mentioned that "we did get license for many software developers to try it out" indicating tangible organizational investment in AI tools. Another highlighted the role of peer influence in organizational adoption, noting "one of the team members tried using [the coding assistant] and he showed us."

Benefit: Democratization and accessibility

Some participants mentioned how they value generative AI's potential in democratizing technical capabilities, making them more accessible. One participant highlighted "the ability to have people that are not experts on programming to actually develop" suggesting that AI tools can enable those without formal programming backgrounds to contribute to development." Another participant captured this as well, describing AI as "a way to make intelligence general, omnipresent, and cheap." [1] This accessibility was seen as particularly valuable in the context of increasingly complex systems, with another participant noting that "things become more complex over time so generative AI could perhaps balance that out a bit."[1] Furthermore, one participant talking about an available generative AI tool mentioned that "It's easy to use. It's integrated to my IDE."

Benefit: AI collaboration

One participant viewed generative AI as more than just a tool, but rather as a collaborative partner in their work. One participant elaborated on this partnership, explaining that "AI is like a way to practice pair programming by yourself." [1] The participant further developed this concept, describing AI as "a whole set of virtual colleagues that make up your team." [1]

This was seen as having the potential to transform the role itself, with one participant noting that "more and more we are moving towards not being developers in the traditional

[1]This excerpt has been translated from Swedish to English

sense but rather being project managers of ourselves with a set of agents that perform the jobs for us where we take on a more and more orchestrating role" [1]

One participant described how generative AI, like human colleagues, can stimulate intellectual growth by creating "opportunities to challenge yourself to be provoked, to be forced to twist and turn concepts that we as humans are extremely poor at doing if we are not forced to do it." [1]

4.3.2 Theme: Code assistance and generation

This theme captures the various ways engineers leverage generative AI to assist with code generation, modification, and understanding in software testing contexts. Participants described multiple benefits related to how AI tools can generate code, provide suggestions, and handle complexity in testing scenarios.

Benefit: Test generation

A minority of participants highlighted the value of generative AI for generating test code across different contexts. One engineer mentioned the benefit for unit testing: "Yeah, I think that is good. For the ones that are standard unit test." The ability to generate tests for existing code was mentioned as valuable in some cases, with one participant explaining: "It can be to advantage if you have legacy-code, which doesn't have any test cases. And you want to make changes to that legacy code. Then it can be nice to create a test case." This suggests that AI tools can help address the common challenge of adding tests to previously untested code that can be difficult or time-consuming to understand.

The wish for AI to support test-driven development was also noted: "this thing with writing test cases first related to test-driven development, when you don't even have any code. How does it work with generative AI? Then maybe you can generate the test cases just based on text?" This reflects how engineers sees the potential in using AI not just to test existing code, but to help in other forms of development as well.

Benefit: Boilerplate and initial setup

Participants consistently highlighted the value of AI for generating some repetitive code structures that are common in testing. One engineer stated that "Boilerplate kind of code or templates can be generated in a faster way" while another specifically noted that "When it comes to testing, there's a lot of boiler-plate code whether you're doing GoogleTest or [...] that I use. There you could use generative AI." The efficiency gains from this capability were emphasized by another participant who said, "I asking genAI to please generate me a template for this unit test and give me like three or four or more test case possibilities." One engineer noted that "all initial setup can be made faster" while another mentioned it was particularly helpful "when you just want to get started because then you have something that compiles quite quickly."[1] Another participant explained, "when a lot of the code would have boiler plate, modules or functions... then the development time is much smaller."

Benefit: Code explanation and suggestions

Participants appreciated generative AI's ability to help them understand code better, which is particularly valuable in testing contexts where engineers often need to work with unfamiliar code. One participant directly stated, "I think it helps me understand the code better. I feel that that's what it is good at." Another engineer mentioned using AI to "generate descriptions, explanations like I want to do this and then you get a description."

Other participants mentioned how they use it to generate suggestions for various tasks. One participant said: "I just use it for suggestions I don't use it like to generate a test and just

[1]This excerpt has been translated from Swedish to English

deploy it." While others more specifically mentioned having names suggested: "Generative AI I think could be great for like naming functions in a good way." and "It's quite nice to have names suggested." [1]

Benefit: Code improvement and refactoring

Beyond generating new code, some participants valued AI's ability to improve existing code. One engineer noted that generative AI "can be good for improving the code, by improving I mean like refactoring." Another mentioned using AI "for refactoring comment et cetera, adding a comment, et cetera, it is very good." The iterative nature of this improvement process was described by one participant: "it's already helpful in writing code like sometimes you're stuck with, you have a good template on what you want to write and you can feed that to the generative algorithm and it can improve on it." Another detailed their workflow as "some kind of iterative process, asking reply, ask, reply, modifying, improving and then at the end it generate like a script that actually feed the purpose of my needs."

Benefit: Handling complexity and edge cases

Some participants emphasized AI's value in addressing complex testing scenarios and identifying edge cases. One engineer noted that "developers are facing complexity when it comes to generating unit test or test cases in more complex test cases that's something we can benefit from using generative AI." The ability to identify overlooked test scenarios was highlighted by one participant who appreciated getting "more edge case testing... because I was not thinks in the first...shot, so I can consider to add more for a specific testing." Another noted the wish for AI to "maybe can identify missing gaps that we have like in our test scope."

Benefit: Focus on higher-level logic

Participants valued how AI assistance with code generation could potentially allow them to focus more on the logical aspects of testing rather than implementation details. One engineer explained, "There's still a lot of typing in programming but it's actually the logic that you really want to focus on. If you can get help with that, definitely I believe in that." The same participant articulated how AI could potentially help testers focus on the testing logic by reducing the complexity: "the actual test logic is the tip of the iceberg, at least for us because we have such complex systems. So there maybe we can get help to reach that peak I think. That we can eliminate a lot of administration."

4.3.3 Theme: Data Analysis

The thematic analysis revealed that participants perceived significant value in generative AI's capabilities for data analysis in a software testing context. This theme contains how generative AI tools can assist engineers in processing, analyzing, and extracting insights from various data sources within their testing workflows.

Benefit: Search and information retrieval

Participants noted that generative AI enhances information retrieval. One participant emphasized the efficiency aspect, stating that generative AI enables "faster search for questions or specific things" and that "If I need to search something, it's easier to just take a question" highlighting the natural language interface as an advantage over traditional search methods.

It was noted by another participant that the versatility of information retrieval, mentioning that "one can ask many questions" about technical topics like Linux commands, indicating

[1] This excerpt has been translated from Swedish to English

that generative AI is used to answer questions the technical instructions like linux commands that often are necessary for software testers to know.

Benefit: Log analysis and troubleshooting

Multiple participants highlighted generative AI's utility in analyzing log files. One participant mentioned that AI can help "check the logs and... give an estimation [of] what could be the fault" suggesting that generative AI can accelerate the identification of issues in test failures. Similarly, another participant specifically stated they "took help of generative AI" to analyze logs during testing activities. This was emphasized further, noting they "use generative AI to improve some areas, data collection, log file analysis" and specifically mentioned "log anomaly detection, log summarization" as areas where testing teams "can benefit quite a lot." The ability to aid in troubleshooting by analyzing logs was seen as beneficial among participants. It was explained by one participant that AI could help "understand based on these logs and these logs and this solution, maybe your issue could be this solution" potentially offering "a smart answer instead of us trying to find in the pile of tickets, something that is similar." Another participant simply but directly confirmed using generative AI for "helping me during troubleshooting."

Participants highlighted how AI tools improve efficiency in finding and processing information relevant to testing activities. One participant noted that "we end up searching ourself in our database without really sometimes knowing what we need to search" and that AI "could help us a lot to reduce time that was spent finding information." The value of AI for analyzing complex information was emphasized by a participant who described using it for test log analysis: "if I have to understand a test log of maybe 1000 lines, and if I have to manually search and understand the logs or any errors, if it takes 30 minutes to do manually, with the help of generative AI, I can wrap up the understanding of logs in 10 minutes." This benefit addresses a key challenge in software testing that is to extract relevant information from large volumes of data efficiently.

4.3.4 Theme: Uncertainty

This theme encompasses the challenges engineers face regarding organizational guidance, security protocols, and knowledge gaps when adopting generative AI for software testing. Participants consistently highlighted tensions between the desire to leverage AI tools for testing and uncertainties about how to do so appropriately within organizational contexts.

Challenge: Security and data confidentiality concerns

Security was a dominant concern that kept reappearing during the interviews, with participants expressing significant uncertainty about what information could safely be shared with AI tools. One participant stated, "I think it's security. For when I ask ChatGPT I check my code that I copy, is there something related to [company name]? If there is anything related to [company name] I will not copy." For engineers working on sensitive components, security concerns were even more pronounced. A participant working in secure development said: "I'm working with the secure port, we are developing secure features, I think it is not okay to use it or to deal with it in my role now." Another participant stated that "the main concern could be sensitivity of the data" highlighting how data security considerations directly impact the feasibility of using AI for certain testing scenarios.

The protection of intellectual property and business secrets emerged as a sensitive area for test generation, where proprietary code must often be shared to generate appropriate tests. One participant noted with concern that responsibility for data security often fell to individual judgment: "Surprisingly, it often becomes individual responsibility even at [company name] that you may not send code to ChatGPT... And let it write test cases, because it's like our business secret." This uncertainty was further expressed by another participant wishing

"to have more clarity on the...of what we can and cannot share in the generative AI tools from [company name]."

Challenge: Lack of organizational guidance

Participants consistently highlighted the absence of clear organizational guidance around AI tool usage for testing purposes. One participant described the current state as a "scattergun approach" elaborating that "within [company name] we have not yet had formal scientific guidance on what has been found to work. So we have effectively ended up with a scattergun approach, everybody is doing something with it." Engineers expressed confusion about who could provide guidance on AI usage for testing: "It's a bit uncertain who to ask. Where to ask? Am I allowed to use this tool? Am I allowed to use this data? Can I use a generated output to some extent? To what extent?." Another participant simply questioned: "If I want to apply this API, who can help me with this?." Several participants called for clearer governance structures and a common way of working. One participant articulated the need for "clearer governance around what you need/should/may do, under what circumstances, and make it very simple so that there is low friction to move forward." Another emphasized that the organization needs to "establish a common way of working, a common framework, common data sources, common action points, common contact persons."

Challenge: Lack of AI knowledge

A knowledge gap regarding AI capabilities and limitations emerged among some participants. One participant observed that "most people only know this generative AI asking questions, but they don't know it can do something more" suggesting limited awareness of AI's potential applications. Another said "it's a little bit difficult for me to understand what the AI will be able to learn and what not."

This knowledge gap extended beyond individual engineers to affect organizational understanding. One participant noted: "There is a lack of knowledge on the potential and especially in the finance organization, there is a lot of concern when it comes to the confidentiality of both numbers, customers and overall data that we are handling." This suggests that knowledge gaps about AI capabilities and limitations affect both technical and non-technical departments. Participants emphasized the need for education before effective adoption could occur: "These generative AI tools need to be learned first. Without that it's no meaning anyone to use generative AI, prompt engineering training. How? How generative AI can be used? What possibilities can use? Only questioning or can be used for pipeline or running anything in a cloud environment?."

Challenge: Transparency

Several participants expressed concerns about the lack of transparency in AI systems used for testing. One participant expressed concerns about the lack of transparency and unclear reasoning behind AI-generated responses: : "When he give me an answer 'I don't know' I will not use it. I don't know how it is trained to answer 'I don't know'." This reflection highlights how the lack of transparency in AI decision-making creates uncertainty in testing contexts, where understanding why a particular test was generated (or not) is often crucial.

Another participant discussing the coding assistant noted: "It's a paid service. So they'll make sure that the main logic is hidden" highlighting how commercial considerations can further reduce transparency in AI systems used for testing. This lack of transparency complicates validation and quality control in software testing, where understanding the purpose of each test is important.

4.3.5 Theme: AI Dependency

This theme captures engineers' perspectives on the essential role of human oversight when adopting generative AI for software testing, and concerns related to overreliance on AI. Furthermore, it captures participants' concerns about the long-term consequences of integrating generative AI tools into testing practices and AI-generated code into code bases.

Challenge: Validation requirements

Participants consistently emphasized the necessity of human validation when generating with AI tools. One participant explicitly stated, "I feel initially instead of blindly trusting the reports, it requires manual validation" highlighting the importance of verification before implementation. Another participant noted that "Currently it is not able to do everything without controls" emphasizing the limited autonomy that should be granted to these tools.

This validation requirement presents a challenge for adoption, as it potentially reduces the efficiency gains that might otherwise be realized through automation. Engineers must balance the time saved in initial test generation against the time required for validation.

Furthermore, participants acknowledged the limitations of AI tools and the responsibility engineers have to validate its output. One participant drew parallels between AI and human abilities to make errors: "Anyone that takes decisions can make a failure, can take a wrong decision. It's the same for us. I mean, we are humans. We can take [wrong decisions]. So it's also for the AI to give us, like, you know, like statements. Okay, this is, you know, this should be that. Of course they can fail." This perspective frames AI as a collaborator in the testing process, sharing the responsibilities with humans. Another participant reinforced this view, stating: "it's also up to you as a person, as a technician to you know, to understand [...] that is also part of the technical work and experience that we do also supporting these tools."

Participants emphasized that effective use of AI for test generation requires substantial domain knowledge and testing expertise. One participant asserted that "when it comes to writing the actual test I think it should still be done by developers, because then you actually know what you are testing" highlighting how domain understanding remains crucial for creating meaningful tests. Another participant elaborated on how human experience provides critical context that AI lacks: "But at the end the experience of the tester complements that because they are working in development or design. They have a lot of background and like okay, we had an issue in this area. We have to take care of this type of interaction or we have to based on experience and that probably the AI is a little bit more difficult to get to understand."This dependency on domain expertise presents a challenge for adoption, as it suggests that AI tools may be less beneficial for less experienced testers or for testing unfamiliar systems which it has not been trained on. One participant explicitly stated that "at this point in time, we would need domain experts in the loop" suggesting that while AI capabilities may evolve, current implementations require human oversight.

Challenge: Skill degradation

Participants expressed concern that overreliance on AI could lead to skill degradation among testing professionals. One participant stated that AI "will maybe make us more stupid" while another shared observations about interns who "hardly know how to code on their own, and therefore they do not know what are the features that they must be asking for right? For a code to work." This suggests that overreliance on AI could lead to testers lacking the foundational knowledge needed to effectively prompt, evaluate, and maintain AI-generated tests. One participant explicitly warned against replacing human knowledge with generative AI: "I think you probably get more disadvantages than advantages if you imagine that you're going to replace knowledge."

Challenge: Increased technical debt and maintainability issues

Several participants expressed concern about the maintainability of code bases containing substantial amounts of AI-generated test code. One participant directly stated, "A concern would be like the technical debt would increase. People might not use it in a good way and then it's more code introduced into the repository that people don't understand." This lack of comprehension creates significant maintenance challenges, particularly when developers need to modify or debug test code in the future.

Some engineers talked about what they expressed as "vibe coding", where developers use AI tools without understanding what the generated test code actually does, emerged as a worrying trend. One participant observed: "A lot of people are using this, without really understanding what is happening... So what you end up with is you risk ending up with code that nobody knows what it does." This concern is made more worrisome by the non-deterministic nature of generative AI, with another participant highlighting that "even if the prompt is the same, it will be impossible to replicate it, so we run the risk of ending up with code bases which will become unmaintainable." This maintainability challenge is particularly significant for test code, which must evolve alongside production code and maintain accuracy over time. As one participant noted, "I spend like 20% of my time actually writing the code 80% of the time is like actually looking into what I'm changing and what it's going to have an impact on. And if people don't have an understanding of how to do that part, I think it's gonna lead to bigger problems in the future."

Challenge: Misapplication of AI in testing processes

The misapplication of generative AI in testing was seen as a significant concern. Several participants viewed the use of AI for generating unit tests as fundamentally misunderstanding the purpose of testing in software development. One participant articulated this clearly: "Unit tests are not a testing tool, it's a development tool and a development methodology. We use the tests to help us think about how we really want the code to behave, not so much to verify it." [1] By automating this process, developers might bypass the critical cognitive work of test design: "Yes, you save time. But what you're removing was perhaps the core of what you were doing in the first place." [1]

Another participant expressed concern about a "conceptual short circuit" when using AI to generate tests from production code, noting that this approach makes a dangerous assumption: "You're making an assumption that the production code you're starting from is actually what you want and that it's somehow correct, because what you're doing is locking it into a kind of specification made up of the tests you generate."[1] This concern highlights how AI-generated tests might create a false sense of security by simply reflecting the existing code rather than independently verifying its correctness. One participant warned that it is "very easy to fall into the trap where we use it for the wrong things and thereby shoot ourselves in the foot."[1]

Challenge: Potential change in review dynamics

An interesting challenge emerged around the review process for AI-generated test code. One participant expressed concern that reviewing AI-generated code might lead to a more passive approach compared to reviewing human-written code: "Everyone has reviewed [code] at some point and I think it's really boring myself. You have to allocate a lot of time for it and really go through it. How does this work? After a while you think, if I had written this myself I would have understood how it works. But then you still have someone who you think has thought about it. But if it's generative AI that you yourself need to review, it's easy to end up in a passive 'this probably works' [mindset]." [1] This suggests engineers may need

[1]This excerpt has been translated from Swedish to English

to develop new review strategies or techniques to maintain the same level of critical analysis when evaluating AI-generated tests.

4.3.6 Theme: Practical integration barriers

This theme addresses the practical challenges engineers face when attempting to incorporate generative AI tools into their software testing workflows.

Challenge: Lack of support and knowledge sharing

Inadequate support for AI adoption was a recurring theme. One participant directly stated: "I felt the support in adopting these tools is not excellent, it can be improved" while another mentioned: "It is difficult getting support for this generative AI. Once I find a program I have to raise a ticket, because there is not everybody spend much time to read ticket." The quality of educational resources was criticized: "these presentations we've attended have been almost ridiculous where you make a prompt like 'you are an expert at making unit tests'... you instruct ChatGPT and then you get some very simple unit test case." [1]

Participants expressed desire for better knowledge sharing mechanisms, with one stating: "I would like to be a part of the forum where I can share the ideas" and another requesting "better tutorials, maybe a quick one minute video in getting started or documentation." Furthermore, one participant expressed: "Maybe you would like there to be a little more local presence?" [1] , with another participant wanting "easier access to experts that can speed up your knowledge."

Challenge: Organizational and resource constraints

Access to advanced models emerged as a significant organizational constraint. One participant described an ongoing ticket process to access more capable models: "I applied for this model, [model name], and I think this model is more smart for instruction. I applied but the ticket is ongoing." This friction hinder engineers' ability to experiment with potentially more effective testing tools.

Resource limitations for customization and fine-tuning were highlighted by multiple participants. One noted: "We would need more GPUs so we can sprinkle them around to everyone who wants to do these kinds of things." [1] Another described attempting to customize an existing tool: "I was trying to replace the LLM which is being used by [the coding assistant] with a different LLM, either a fine-tuned one or a different like [model name]. I was not able to do it." The cost implications of training models were explicitly mentioned by an participant: "It did involve additional resources to take out the pre-training, especially if I want to use an efficient training where I increase the rank, increase the trainable parameters. It will be an expensive factor."

Challenge: Token limitation and prompt engineering

Participants frequently cited some limitations related to LLMs as significant obstacles. Token limitations emerged as a particularly frustrating constraint, with one participant explaining: "I think for one big problem is they have some token limitation... generally I need several rounds to make it understand and make it answer my question. But generally when I try 2 or 3 rounds no answer, because this token limit." This limitation is especially problematic for testing scenarios that require substantial context about the system under test.

The iterative nature of prompt engineering for test generation was also identified as inefficient: "it takes ten prompts to get a working script, if it is reduced to two or three, it would be better." Similarly, another participant noted the need for detailed specifications in prompts:

[1] This excerpt has been translated from Swedish to English

"if I not specify how unit tests need to be tested, so what need to be tested and assert in the end, the code block was not knowing what I would like to test."

Challenge: Diverse user needs

A significant challenge for organizational AI adoption is addressing the spectrum of user needs. As one participant said: "there are so many different types of usage as well, there are like power users who want GPU clusters to be able to fine-tune models for their specific cases and then there are your 'enterprise users', who thinks AI is just equal to ChatGPT, and a spectrum in between." [1]

This diversity creates tension in resource allocation and support strategies, with one participant noting: "he said, he want balance between all the people. For he don't want one person to use much."

4.3.7 Theme: Accuracy and reliability of AI output

The quality and reliability of AI-generated test code emerged as a critical theme in participants' experiences with generative AI for software testing.

Challenge: Output quality and reliability

Participants consistently reported experiencing unpredictable quality in AI-generated tests, with reliability varying significantly between interactions. One participant candidly stated, "To be honest not so much. I tried to use it before, and most times it works well and sometimes not. It like stopped or something." This inconsistency undermines confidence in the tools and creates uncertainty in testing processes where reliability is paramount.

The mixed reliability as particularly problematic in the context of test generation, where engineers need dependable outputs to build robust test suites. One participant described their experience: "sometimes when you ask question and get a part of c-code or something. Sometimes you took a lot of time into try to debug it or to see what is the problem so sometimes it is not so efficient." This suggests that while AI might accelerate initial test creation, the debugging overhead can sometimes negate these efficiency gains. This highlights a crucial consideration for testing workflows where correctness is essential.

Multiple participants reported that AI tools would sometimes fail to respond appropriately, with one stating, "sometimes he answer me I don't know, or sometimes nothing happens, no answer."Another noted, "Sometimes it stuck or something if you give a question that its not okay for it to answer."These failure modes create friction in the testing process and can lead engineers to abandon the tools entirely, as evidenced by the statement: "when he [the model] give me an answer 'I don't know' I will not use it."

Challenge: Complexity in software testing

The reliability of AI-generated tests appeared to decrease as the complexity of the testing scenario increased. One participant observed: "But if it [using generative AI] is for something little more complex, it generates output but is generally useless and very hard to get it to fix it." This limitation poses significant challenges for software testing, particularly when dealing with complex component interactions in dependency-heavy code bases and when attempting to identify and verify edge cases that require deep system understanding.

Another participant highlighted the difficulty in generating tests from requirements: "unit tests where we have this shift-left strategy where you should test more at the unit-test and bit level. There you want to start from these requirement sequences in that case... And it feels like it would place very high demands on the requirements. And it might be that you have

[1] This excerpt has been translated from Swedish to English

to sit and specify them so rigorously that it wouldn't provide any benefit." [1] This observation highlights a fundamental challenge in test generation which is that the specificity required to generate accurate tests might sometimes exceed the effort of writing the tests directly, particularly for complex requirements.

The complexity and large scope of testing domains presented another technical challenge. As one participant noted, "when you add more sophisticated code then like a whole new framework, maybe that is a little bit more tricky to understand how to test it."

Challenge: Domain knowledge limitations

One participant noted that the generic training of most AI models limited their effectiveness for domain-specific testing scenarios. The participant suggested that "if it was trained with [company name]'s data itself, it could have given a better response" indicating that domain adaptation is thought of as particularly important for effective test generation. The lack of domain-specific knowledge creates challenges for testing proprietary systems or applying company-specific testing practices.

The domain adaptation challenges were further explained: "there are certain terms which is proprietary to [company name]. Now if I have to take any help from generative AI, it is not trained with this proprietary tool in terms of the product. So there I face challenge." This limitation is particularly relevant for test generation, where domain-specific knowledge is often necessary.

Table 4.9: Benefits perceived by participants ($N = 12$), and the number of participants that talked about each

Benefit	**# participants**
Time savings and efficiency improvement	11
Organizational support and encouragement	9
Democratization and accessibility	3
Test generation	3
Boilerplate and initial setup	3
Code explanation and suggestions	3
Log analysis and troubleshooting	3
Search and information retrieval	3
AI collaboration	2
Code improvement and refactoring	2
Focus on higher-level logic	1
Handling complexity and edge cases	1

4.4 Summary of thematic analysis

The thematic analysis revealed a total of 9 themes, which contributed to 12 identified benefits and 16 identified challenges. To summarize our most important findings in response to RQ2, we present the benefits and challenges discussed by the most participants in the interviews, as shown in Table 4.9 and Table 4.10, respectively.

From Table 4.9, two benefits stands out as being mentioned the most, namely *Time savings and efficiency improvement* and *Organizational support and encouragement*. Eleven out of the twelve participants acknowledged tasks related to software testing where they saw a significant reduction in time spent spent solving them. Furthermore, nine participants expressed the organizational support they received when adopting generative AI tools as beneficial. The third most mentioned benefit was only expressed by three participants. From this result

[1] This excerpt has been translated from Swedish to English

Table 4.10: Challenges perceived by participants ($N = 12$), and the number of participants that talked about each

Challenge	**# participants**
Security and data confidentiality concerns	6
Validation requirements	6
Output quality and reliability	5
Lack of support and knowledge sharing	5
Lack of organizational guidance	4
Lack of AI knowledge	4
Complexity in software testing	3
Token limitation and prompt engineering	3
Organizational and resource constraints	3
Skill degradation	3
Diverse user needs	2
Transparency	2
Increased technical debt and maintainability issues	2
Misapplication of AI in testing processes	2
Domain knowledge limitations	1
Potential change in review dynamics	1

we acknowledge a potential flaw in our interview procedure, where two of the questions were very closely connected to these two benefits, which could have attributed to these being mentioned the most. However, the participants explicitly identified these as benefits rather than challenges. While our interview procedure may have slightly inflated the number of mentions compared to other benefits, the fact that participants viewed them as beneficial is still valid and supported by our data. Both of these themes were also talked about in relation to other interview questions, so we believe they would have been brought up regardless. We also note that these benefits are not always seen as beneficial in all contexts. For example, participants acknowledged that it is mainly for simpler tasks such as boilerplate code generation that they notice the efficiency improvements. Testers frequently mention the validation requirements sometimes making them spend more time fixing or debugging AI-generated code, than it would have taken them to write the code themselves. In addition, while the organizational support is seen as beneficial by most, there is still some aspects of it that presents challenges, as reflected by the challenge *Lack of support and knowledge sharing*.

The number of participants for each challenge is more evenly distributed, as seen in Table 4.10. The table shows *Security and data confidentiality concerns* and *Validation requirements* being the most frequently mentioned challenges with six participants each. To answer RQ2 we also include the challenge *Output quality and reliability* as there were only one less participant (five) seeing this as a challenge. We exclude *Lack of support and knowledge sharing* as being one of the most prominent challenges as the closely related benefit *Organizational support and encouragement* is mentioned more frequently.

Answer to RQ2

The thematic analysis resulted in 7 themes, revealing 12 benefits and 16 challenges. The most prominent benefits were: *Time savings and efficiency improvement* and *Organizational support and encouragement*, while the most prominent challenges were: *Security and data confidentiality concerns*, *Validation requirements*, and *Output quality and reliability*

5 Discussion

This chapter discusses the results and the methodology, and concludes by placing our work within a broader context.

5.1 Results

In this section, the most important findings are investigated along with their implications.

5.1.1 Compilation success rate

The compilation success rate seems a bit low considering that Yang et al. [51] reach a compilation success rate between 38.17% and 52.96% for six different LLMs. However, the narrow focus of the thesis restricts the ability to generalize the findings over different domains and implementations. As mentioned by Yang et al. [51], generating syntactically correct tests remains a challenge. From the perspective of a company, it is not feasible to use a coding assistant to automatically generate unit tests when there is a larger probability that the test is not compilable than that it is. This need of output reliability is also a challenge found in the user study. One solution to this could be involving a human to collaborate with the coding assistant by validating its output. However, overseeing each generated unit test does not only come with increased resource costs for the company, but might also be slower than the human writing the test from scratch as it removes the need to understand the generated test in order to find any mistakes in the code. Instead, a more economical solution could be to execute the tests in a similar manner as in this work. This way the company could choose to instantly throw away generated tests that are not compilable or that do not pass. While executing tests does use company resources, executing the generated tests before deciding whether or not to keep them would only add a non-recurring cost.

Takeaway 1: The quality evaluation revealed that generating compilable unit tests is a key issue for the coding assistant, with over half of all generated tests failing to compile. Companies interested in adopting such tools should expect reliability challenges, though broader studies are needed to generalize this finding.

5.1.2 Errors

The errors found were unexpected as the coding assistant uses RAG, meaning it should be able to extract relevant code snippets and thus use the knowledge of what names have been defined to avoid such errors. According to Yang et al. [51], this issue can be attributed to LLMs' tendency to hallucinate. As the company's code base is large and complex, the coding assistant's ability to extract relevant context is likely too limited to combat such hallucinations. Yang et al. [51] evaluated EvoSuite, which conversely does not use an LLM, and found a much higher compilation success rate of 85.71%. However, they underscore issues regarding the readability of tests generated by the tool. This means that there is currently a trade-off between compilability and readability in the field of automatic unit test generation. Prior research has investigated the use of search-based test generation tools like EvoSuite, followed by LLM post-processing to enhance test readability while maintaining test semantics [6][41]. Although this approach shows promise, the company where this thesis was conducted wanted to evaluate the applicability of a commercial coding assistant (already integrated into their development environment) for unit test generation, rather than developing a new specialized tool.

Takeaway 2: We see that the usage of undefined keywords is the main cause of errors when generating unit tests using a coding assistant. This is likely due to the coding assistant's inability to extract relevant context in a complex industrial code base.

5.1.3 Test length and similarity

The strong correlation between shorter test lengths and higher quality means that companies should begin with limiting the application of LLM-based automatic unit test generation to shorter methods under test before moving on to longer ones. While this would reduce the amount of unit tests that are generated automatically, maintaining a high quality test suite is very important for companies, especially in the telecommunications domain. Companies wanting to transition to automatically generating unit tests should consider coding practices with short class methods, as our results show that a shorter MUT length correlates to generated tests of higher quality.

Takeaway 3: We find that generated tests for shorter methods under test have higher quality. Companies should focus on automating unit test generation for shorter methods under test before moving on to longer ones.

The results show that generated tests with lower number of lines have a non-zero compilation success rate. This suggests that attempting to regenerate tests within such a length limit may yield a higher compilation success rate. A future solution could be to repeatedly provide feedback from the compiler to the LLM, and then ask it to correct the test based on the feedback. Longer tests may not be worth regenerating due to their lack of successful compilations apart from outliers.

While both the mean and median similarity scores were in the low resemblance range for each of the similarity metrics for the tests that did not compile and pass, the significantly higher similarity among Category I tests further reinforces the idea of filtering out all tests outside of this category. The higher Levenshtein similarity for Category I tests suggests that such filtering would add tests that likely require little modification to follow the company's coding practices, which is important in order to avoid a reduction in maintainability.

5.1.4 User study

The user study explored engineers' perceptions of generative AI tools in software testing, focusing on both the benefits and challenges encountered in practice. Participants shared insights on efficiency gains, quality limitations, evolving developer roles, and organizational responsibilities related to AI adoption.

Efficiency and Quality Challenges *Time savings and efficiency improvement* emerged as a prominent benefit in the interviews. However, this benefit was primarily observed in simple, code-related software testing tasks, such as generating initial boilerplate code, comments, name suggestions, and other tasks not related to coding such as search and data analysis. Participants acknowledged a complexity threshold in software testing beyond which the quality of AI-generated output is insufficient for producing accurate results. This observation aligns with the findings of Klemmer et al. [27], Coutinho et al. [13], and Liang et al. [30], who also noted developers face challenges with output quality that diminish time-saving benefits. Our study's participants also pointed out that the need to validate and debug AI-generated output sometimes outweighs the initial efficiency gains, a challenge also identified by Coutinho et al. [13], Hamza et al. [22], and Liang et al. [30]. The implication for software testing scenarios is that, given the current state of generative AI tool quality, different approaches should be explored to enhance software testing tasks. One participant mentioned that generating tests by displaying source code is a misapplication of the technology, and another suggested to instead explore test-driven development to generate tests from natural language descriptions. This indicates there are ideas among software testers for advancing the exploration of generative AI's applicability in software testing.

> **Takeaway 4:** Engineers perceive generative AI as increasing their efficiency in initial phases of software testing tasks, but acknowledge the loss in quality during complex tasks and the accompanying need of validating its output.

Future Roles and Skill Implications Another finding is the expectation among participants that AI will eventually enable developers to transition into more orchestrating roles, similar to the findings of Klemmer et al. [27]. This shift could reduce the administrative burden on developers working in complex systems, allowing them to concentrate on higher-level logic. However, participants acknowledged the current necessity of validating and critically analyzing AI-generated output given the present quality of generative AI tools. This suggests that such a transition might take time, and human oversight should remain crucial during this potential shift. Participants also expressed concerns about overreliance on generative AI and skill degradation, which could pose obstacles during the transition. Organizations should therefore emphasize skill development, enhancing the ability to critically review AI-generated code and limit potential increases in technical debt, as feared by our participants.

Organizational Guidance and Security Concerns Finally, although we recognize the company's efforts to provide usage guidelines and empower engineers with generative AI tools, uncertainty remains regarding safe usage guidance. Discussions about security concerns revealed challenges in using these tools due to risks of leaking proprietary code or business secrets, similar to findings by Liang et al. [30] and Klemmer et al. [27]. Therefore, organizations should enhance efforts to guide users more effectively and safely through education and guidelines.

Takeaway 5: Engineers predict AI will shift developers toward more orchestrating roles but stress the need to validate AI-generated work. They worry about skill loss and technical debt from overreliance on generative AI and urge companies to offer clear usage guidelines and skill development opportunities.

5.2 Method

This section discusses the limitations of the methods used in the quality evaluation and the user study.

5.2.1 Black-box coding assistant

Due to the black-box nature of the commercial tool, it is difficult to give an accurate representation of the exact methodology used. Firstly, while the general structure of the LLM is known, the exact model is proprietary. Consequently, some important information such as model context window, model architecture and model size remains unknown. Secondly, the coding assistant gathers relevant information from the relevant files, and adds it to the prompt. It is unclear exactly what the prompt includes, making it hard to compare the approach to other research where the prompt tends to be a central part of the described methodology. Lastly, there is no guaranteed consistency in the output of the model. Providing the assistant with the same exact prompt twice might lead to two completely different generated unit tests, one not even compiling while the other compiles and passes. This limits the reproducibility of our results, even if access to our dataset was provided.

5.2.2 Industrial code base

There are some limitations in terms of the collected dataset. Our mapping between methods under test and their corresponding tests relies solely on detecting direct function calls from the test to the MUT, as described in Section 3.1. This approach may miss valid test-method pairs where the relationship is more indirect, such as when a test calls a MUT through intermediate functions or dependency injection. Additionally, simply detecting a function call doesn't guarantee that the test is primarily focused on testing that particular method - it might be calling it incidentally as part of testing something else. Without expert knowledge of the codebase or formal test documentation, we cannot verify the completeness or precision of our mappings. This may result in both false negatives (valid test-method pairs we fail to identify) and false positives (incorrectly matched pairs) in our dataset.

Furthermore, the data is specific to one module in the telecommunications industry, which could be either under-represented or over-represented in the training data used for pretraining the used LLM. As described above, the model context window is unknown, making it difficult to exclude datapoints that are too large. If the datapoint size limit is set too high, it affects the model's ability to use the whole input in order to generate the output. Conversely, if the limit is set too low, relevant datapoints might be excluded, with a smaller dataset making it harder to generalize findings because it may not capture the full variability present in the code base, especially since this exclusion targets the specific group of large datapoints, making the average size of datapoints significantly smaller.

It is not only the dataset that is affected. As the coding assistant generates an index of the code base, and uses the relevant context in the prompt, one would need a nearly identical code base in order for the coding assistant to extract similar relevant snippets, which is crucial for generating similar output.

The use of single prompting is another limitation of the method. As opposed to multi-turn prompting, single prompting does not allow for iterative improvements by providing feedback on the generated output. Instead, it requires the initial prompt to contain all the

relevant information. As finding all the relevant information for a unit test is not an easy task, using a multi-turn prompting approach could yield better results.

5.2.3 Participants

The participants recruited for the user study had a diverse range of experience with software testing and generative AI usage. Some participants held current roles related to software testing, while others had more experience with software testing in the past. Although all participants had experimented with generative AI to some extent, some had not directly applied generative AI to software testing tasks. This presents a validity threat, as their perceived benefits and challenges may be based more on their general experience with generative AI rather than its specific application to software testing. To address this threat, the interview structure incorporated branching logic, with certain questions tailored to participants who had not used generative AI in conjunction with software testing directly.

Additionally, reasons for non-adoption often remain unexplored in many adoption studies [26][42], which our study aims to address. We also acknowledge that the sample size of 12 participants is small and predominantly consists of senior engineers (mean career length of 15 years) working in a large, safety-critical domain. The company culture may therefore influence perceptions of the benefits and challenges, such as a strong security culture heightening concerns related to security.

5.2.4 Interviews

Interviews were chosen over surveys as the preferred data collection method for the user study because they allowed the interviewers to ask participants for clarification and delve deeper into the reasoning behind their answers, which is not possible with surveys. The study aimed to achieve greater depth rather than the breadth that a survey-based approach would provide. The interview guide was drafted based on constructs from TAM, DoI, and UTAUT. Anchoring the study in these well-established theoretical frameworks ensured wide coverage of adoption-related factors known from prior literature. This approach was intended to mitigate the limitation stemming from the authors' limited experience with software testing practices, interviews, and qualitative data analysis. For each interview, the author who did not conduct the interview was responsible for transcribing it, followed by proofreading by the other author. This simple cross-checking process helped reduce transcription errors early on and provided each researcher with an initial, in-depth exposure to the raw data, thereby increasing familiarization before coding began.

5.2.5 Thematic analysis

Braun and Clarke's six-phase procedure [8] was followed by a deductive orientation. The deductive approach kept the analysis closely aligned with RQ2 and ensured that the trail from excerpt $\rightarrow$ code $\rightarrow$ theme, was clearly explained. This clarity allows for replication of this part of the methodology in similar studies.

Though we followed established guidelines from Boyatzis [7] and Braun and Clarke [8] for segmenting the transcriptions, the process of deciding where excerpts begin and end remains subjective. This subjectivity could affect reliability since different researchers might divide the same data differently. Our decisions about excerpt boundaries potentially influenced which benefits and challenges became visible in our analysis. While we tried to capture single concepts in each excerpt, determining the boundaries involves interpretation, which is a limitation of our approach to data preparation.

Both authors were exposed to all the data before engaging in the coding process, which strengthened the reliability of the subsequent analysis. The process of coding the excerpts has some reliability concerns. The initial coding was performed by one of the authors who

was involved in conducting and transcribing the interviews. Each code was reviewed by the other author, and disagreements were resolved through discussions, which enhances the reliability of the results. The shared exposure to the data facilitated more meaningful discussions during code reviews, as both authors could reference the full context of interviews when resolving coding disagreements. However, because of the homogenous background of the authors, there could have been some increased risks of bias being introduced in the procedure. Including a third coder with no prior connection to the study could have improved reliability by providing an outside perspective that was not influenced by our involvement in data collection or our shared assumptions. Furthermore, we did not compute any inter-rater reliability statistics during the coding procedure, consequently omitting any quantifiable metrics for reliability.

As a final remark, because the company operates under strict NDAs, full transcripts cannot be released. This limits the extent to which external researchers can replicate the entire thematic analysis and draw their own conclusions. As a compromise, we provide a rich sets of excerpts in the thematic analysis in Section 4.3.

5.2.6 Source criticism

To ensure the credibility of this work, the sources used mostly consist of peer-reviewed journals. The most common publishers used are ACM and IEEE, due to the their reputation of having a high standard for publishing. In some cases, articles were not published in any journal. In these cases, only works that contain highly relevant information and have been cited by plenty of other peer-reviewed work were used, and even then with skepticism.

5.3 The work in a wider context

With regards to the use of generative AI in software testing, we identify some relevant ethical and societal aspects.

The first one is the environmental effects of generative AI. While there is plenty of research on the energy consumption of training LLMs [23], our concern is instead the energy consumption and consequent carbon emissions that result from the inference when using generative AI at an increasingly large scale. While prompting a coding assistant one time does not have a large energy consumption, it scales with the number of users as well as with the frequency of the usage. We believe that generating tests automatically using generative AI does not have a large environmental impact as it is a single generation for each test. However, if generative AI introduces redundant tests in the codebase, the environmental impact comes from repeatedly executing these tests throughout the software's lifecycle. The cumulative energy consumption of running unnecessary tests thousands of times across an organization can lead to significant energy consumption, especially in large industrial settings where test suites run frequently. In addition, the rapid adoption of generative AI in other fields in software engineering as well as in the daily life of the everyday person poses a climate risk that needs to be adressed.

The effect of automation on employment rates is another concern. As generative AI continues to improve, it is able to replace more of human labour, leading to a rapid rise in unemployment. While this change is likely to create new jobs as well, it is uncertain whether these can make up for the jobs lost. The thematic analysis shows that, for now, generative AI is not reliable enough to completely replace humans in software testing. Instead, it appears to be a powerful assisting tool, increasing our productivity. The models are still improving and it remains to be seen to what extent they will replace the human workforce.

6 Conclusion

This chapter provides answers to each of the research questions stated in Section 1.3. It concludes with us presenting aspects of our work that can be extended upon for future work.

6.1 Overview

The purpose of this thesis was to evaluate a commercial coding assistants in terms of automatically generating unit tests in an industrial software testing environment, and to explore how engineers working in such an environment perceive the adoption of generative AI tools for software testing tasks. Together these investigations aim to give an overview of the current capabilities of applying generative AI to software testing tasks in industrial settings, both from a quantitative perspective and from a qualitative one. By conducting an evaluation to measure the quality of a generative AI tool in this context, and a user study with engineers to reveal their perceptions, our research contributes to understanding how the technology performs versus how it is perceived, within a software testing context, and serves as guide for companies seeking to evaluate the adoption of generative AI tools, especially for automatic unit test generation.

RQ1: How is the quality of unit tests generated by a commercial generative AI tool in an industrial context?

RQ1 aimed to evaluate the quality of unit tests generated by a coding assistant on the code base of a telecommunications company. We assessed 12,000 generated unit tests for compilation success rate. Around one-third compiled, with one-fourth compiling and passing. Comparing these tests to gold standard equivalents using cosine, Jaccard, and Levenshtein similarity measures revealed generally low resemblance. However, the resemblance significantly increased for those tests that compiled and passed. The findings suggest that the coding assistant struggles with generating compilable unit tests. Similar tools should therefore be adopted without expectations of high reliability. However, a broader study would provide more generalizability to the findings. Additionally, companies seeking to automate unit test generation should focus on generating unit tests for short methods under test, and should execute and evaluate automatically generated tests before considering adding them to the code base so that low-quality tests are likely to be filtered out.

RQ2: What are the benefits and challenges of adopting generative AI tools for software testing as perceived by engineers?

RQ2 explored the perceived benefits and challenges of generative AI tools in software testing. While participants appreciated the empowerment these tools provide, they raised concerns about data confidentiality and inadequate usage guidelines, emphasizing the need for companies to provide better guidance. Engineers also worried about increased technical debt and skill degradation, we therefore suggest organizations to educate about critically reviewing AI-generated code. Although AI tools were seen as efficient for simpler tasks, validation requirements and quality issues limited their effectiveness in complex scenarios. Participants were cautious about using AI for generating unit tests, viewing it as a potential misapplication, suggesting organizations should explore alternative strategies to enhance benefits for software testers.

6.2 Future work

This work could be expanded in several directions to deepen the understanding of generative AI's application to software testing in industrial environments. This holds both regarding the quality evaluation as well as for the user study.

Quality evaluation An area worth exploring is the performance of the generative AI tool on additional code bases within the company. This approach might reveal whether particular code characteristics make certain code bases more amenable to automated test generation than others, potentially leading to more targeted application guidelines. Since the internal LLM of the coding assistant, had not been fine-tuned on the proprietary code base, an area for exploration is how replacing this generic model with a fine-tuned LLM would affect the results. This possibility was highlighted in our user study, where participants suggested that fine-tuning could increase the appropriateness of applying generative AI to proprietary code bases. Another aspect worth exploring is the impact of prompt engineering on output quality. Our current study employed a single prompting approach, but a comparison of various prompting techniques might yield significant improvements in the quality of generated tests. A comparative evaluation of different commercial and open-source generative AI tools on identical testing tasks would provide valuable benchmarking data for organizations considering adoption. Such research could identify which tools perform best for specific testing scenarios and code base characteristics.

User study Future research would benefit from recruiting participants who specifically work in software testing roles and actively use the same generative AI tool that we evaluated. This would provide more targeted insights into the perceived benefits and challenges for tasks directly related to software testing practices. Additionally, including participants who regularly use the specific tool being evaluated would enable a more focused analysis of user perceptions. Since all participants in our study were from the same company, conducting a cross-company study would help mitigate the influence of company-specific culture, potentially leading to more generalizable findings about engineers perceptions. Furthermore, longitudinal studies tracking how teams integrate these tools into their workflows over time would reveal how perceptions evolve over time and which initial challenges are overcome through experience, and which persist. Finally, investigating how these tools can be most effectively integrated with existing testing workflows could address many of the practical integration barriers identified in our user study. This could lead to the development of more comprehensive guidelines for the effective integration of generative AI in industrial software testing practices.

Bibliography

[1] Icek Ajzen. "The theory of planned behavior". In: *Organizational Behavior and Human Decision Processes* 50.2 (1991). Theories of Cognitive Self-Regulation, pp. 179–211. ISSN: 0749-5978. DOI: 10.1016/0749-5978(91)90020-T.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "code2vec: learning distributed representations of code". In: 3.POPL (2019). DOI: 10.1145/3290353. URL: https://doi.org/10.1145/3290353.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[4] Albert Bandura. "Social cognitive theory of moral thought and action". In: *Handbook of Moral Behavior and Development*. Psychology Press, 2014, pp. 69–128.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. *Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code*. 2022. arXiv: 2206.01335 [cs.SE].

[6] Matteo Biagiola, Gianluca Ghislotti, and Paolo Tonella. *Improving the Readability of Automatically Generated Tests using Large Language Models*. Dec. 2024. DOI: 10.48550/arXiv.2412.18843.

[7] Richard E. Boyatzis. "Transforming Qualitative Information: Thematic Analysis and Code Development". In: 1998.

[8] Virginia Braun and Victoria Clarke. "Using thematic analysis in psychology". In: *Qualitative Research in Psychology* 3 (Jan. 2006), pp. 77–101. DOI: 10.1191/1478088706qp063oa.

[9] Cecilia Ka Yuk Chan and Wenjie Hu. "Students' voices on generative AI: perceptions, benefits, and challenges in higher education". In: *International Journal of Educational Technology in Higher Education* 20.1 (2023), p. 43. ISSN: 2365-9440. DOI: 10.1186/s41239-023-00411-8.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such,

Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL].

[12] Vincenzo Corso, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. "Assessing AI-Based Code Assistants in Method Generation Tasks". In: *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2024, pp. 380–381. DOI: 10.1145/3639478.3643122.

[13] Mariana Coutinho, Lorena Marques, Anderson Santos, Marcio Dahia, Cesar França, and Ronnie de Souza Santos. "The Role of Generative AI in Software Development Productivity: A Pilot Case Study". In: *Proceedings of the 1st ACM International Conference on AI-Powered Software*. AIware 2024. Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 131–138. ISBN: 9798400706851. DOI: 10.1145/3664646.3664773.

[14] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. "Effective test generation using pre-trained Large Language Models and mutation testing". In: *Information and Software Technology* 171 (2024), p. 107468. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2024.107468.

[15] Fred D. Davis. "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology". In: *MIS Quarterly* 13.3 (1989), pp. 319–340. ISSN: 02767783, 21629730. DOI: 10.2307/249008. (Visited on 2025-05-08).

[16] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2025. arXiv: 2412.19437 [cs.CL].

[17] Luciano da F. Costa. *Further Generalizations of the Jaccard Index*. 2021. arXiv: 2110.09619 [cs.LG].

[18] Martin Fishbein and Icek Ajzen. *Belief, Attitude, Intention, and Behavior: An Introduction to Theory and Research*. Reading, MA: Addison-Wesley, 1975. ISBN: 0201020890.

[19] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179.

[20] Greg Guest, Arwen Bunce, and Laura Johnson. "How many interviews are enough? An experiment with data saturation and variability". In: *Field Methods* 18.1 (2006), pp. 59–82. DOI: 10.1177/1525822X05279903.

[21] Rishin Haldar and Debajyoti Mukhopadhyay. *Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach*. 2011. arXiv: 1101.1232 [cs.IT].

[22] Muhammad Hamza, Dominik Siemon, Muhammad Azeem Akbar, and Tahsinur Rahman. "Human-AI Collaboration in Software Engineering: Lessons Learned from a Hands-On Workshop". In: *Proceedings of the 7th ACM/IEEE International Workshop on Software-Intensive Business*. IWSiB '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 7–14. ISBN: 9798400705717. DOI: 10.1145/3643690.3648236.

[23] Geerd-Dietger Hoffmann and Verena Majuntke. "Improving Carbon Emissions of Federated Large Language Model Inference through Classification of Task-Specificity". In: *SIGENERGY Energy Inform. Rev.* 4.5 (Apr. 2025), pp. 44–50. DOI: 10.1145/3727200.3727208.

[24] "IEEE Standard for Software and System Test Documentation". In: *IEEE Std 829-2008* (2008), pp. 1–150. DOI: 10.1109/IEEESTD.2008.4578383.

[25] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL].

[26] Young Woo Kim, Min Chul Cha, Sol Hee Yoon, and Seul Chan Lee and. "Not Merely Useful but Also Amusing: Impact of Perceived Usefulness and Perceived Enjoyment on the Adoption of AI-Powered Coding Assistant". In: *International Journal of Human–Computer Interaction* 41.10 (2025), pp. 6210–6222. DOI: 10.1080/10447318.2024.2375701.

[27] Jan H. Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton, Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, Awais Rashid, Alena Naiakshina, and Sascha Fahl. "Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns". In: CCS '24 (2024), pp. 2726–2740. DOI: 10.1145/3658644.3690283.

[28] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 919–931. DOI: 10.1109/ICSE48619.2023.00085.

[29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[30] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3608128.

[31] Rajiv Movva, Sidhika Balachandar, Kenny Peng, Gabriel Agostini, Nikhil Garg, and Emma Pierson. "Topics, Authors, and Institutions in Large Language Model Research: Trends from 17K arXiv Papers". In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 1223–1243. DOI: 10.18653/v1/2024.naacl-long.67.

[32] Gene Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *J. ACM* 46.3 (May 1999), pp. 395–415. ISSN: 0004-5411. DOI: 10.1145/316542.316550.

[33] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[34] Ciprian Paduraru, Alin Stefanescu, and Augustin Jianu. "Unit Test Generation using Large Language Models for Unity Game Development". In: *Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games*. FaSE4Games 2024. Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 7–13. ISBN: 9798400706745. DOI: 10.1145/3663532.3664466.

[35] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software". In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 15–26. ISBN: 9781605580500. DOI: 10.1145/1390630.1390635.

[36] Muhammad Qorib, Geonsik Moon, and Hwee Tou Ng. "Are Decoder-Only Language Models Better than Encoder-Only Language Models in Understanding Word Meaning?" In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 16339–16347. DOI: 10.18653/v1/2024.findings-acl.967.

[37] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. "Improving language understanding by generative pre-training". In: *OpenAI* (2018).

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners". In: *OpenAI* (2019). (Visited on 2024-11-15).

[39] Rudolf Ramler and Klaus Wolfmaier. "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost". In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 85–91. ISBN: 1595934081. DOI: 10.1145/1138929.1138946.

[40] Everett M. Rogers. *Diffusion of Innovations*. Fourth. Simon & Schuster, 2010. ISBN: 9781451602470.

[41] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. "DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, pp. 287–298. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416622.

[42] Daniel Russo. "Navigating the Complexity of Generative AI Adoption in Software Engineering". In: *ACM Trans. Softw. Eng. Methodol.* 33.5 (June 2024). ISSN: 1049-331X. DOI: 10.1145/3652154.

[43] Gerard Salton and Michael McGill. *Introduction to modern information retrieval*. New York, NY: McGraw-Hill, 1983, p. 121. ISBN: 9781783303229.

[44] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105. DOI: 10.1109/TSE.2023.3334955.

[45] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 121–125. DOI: 10.1109/MSR.2019.00028.

[46] Anselm L Strauss and Juliet Corbin. "Open coding". In: *Social research methods: A reader* (2004), pp. 303–306.

[47] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. *Unit Test Case Generation with Transformers and Focal Context*. 2021. arXiv: 2009.05617 [cs.SE].

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[49] Viswanath Venkatesh, Michael G. Morris, Gordon B. Davis, and Fred D. Davis. "User Acceptance of Information Technology: Toward a Unified View". In: *MIS Quarterly* 27.3 (2003), pp. 425–478. ISSN: 02767783. (Visited on 2025-05-29).

[50] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 365–381. ISBN: 978-3-540-31980-1.

[51] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. "On the Evaluation of Large Language Models in Unit Test Generation". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1607–1619. ISBN: 9798400712487. DOI: 10.1145/3691620.3695529.

[52] Li Zhong, Zilong Wang, and Jingbo Shang. *Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step*. 2024. arXiv: 2402.16906 [cs.SE].