MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

AI-driven Software Development Source Code Quality

Master's Thesis

BC. PETR KANTEK, B.SC.

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

AI-driven Software Development Source Code Quality

Master's Thesis

BC. PETR KANTEK, B.SC.

Advisor: Bruno Rossi, PhD

Department of Computer Systems and Communications

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Petr Kantek, B.Sc.

Advisor: Bruno Rossi, PhD

Acknowledgements

I would like to express my gratitude to Dr. Bruno Rossi, my supervisor, for the numerous constructive feedback sessions and insightful discussions about the thesis. I would also like to thank my family for everything.

Abstract

In recent years, applications of artificial intelligence have seen notable success across various fields. Large Language Models (LLMs) have particularly found extensive use in the field of software development. From source code generation and code understanding to documentation translation, tools based on LLMs can enhance the effectiveness of the software development life cycle and assist software engineers in their daily tasks, all within the comfort of their favorite Integrated Development Environments (IDEs). An open question regarding these tools revolves around the quality of the source code they produce, as low-quality code could potentially do more harm than good, especially when it comes to common security vulnerabilities in source code.

Building upon this line of thought, this work establishes two key goals. The first goal is to identify the best AI-based tool from a selected list of popular options, including GitHub Copilot, Tabnine, ChatGPT, and CodeGeex, for source code generation based on the quality of the produced code. The experiments reveal that GitHub Copilot takes primacy, demonstrating the best results across eight research questions. The second goal involves deploying a custom LLM tool for code generation using the TabbyML platform and cloud hosting. This custom tool is then integrated with IDEs.

Keywords

Large Language Models, Transformer, Deep Learning, Static Code Analysis, Software Quality, Vulnerabilities

Contents

List of Tables

List of Figures

Introduction

Artificial Intelligence (AI) has become increasingly prominent in various domains. One area where AI holds immense potential is in software development. AI-based tools can instantly generate source code across a wide range of programming languages, enhancing developers' effectiveness throughout the software development life cycle. As organizations aim to deliver high-quality software products within shorter time frames, ensuring source code quality becomes especially crucial when incorporating AI-generated code into the codebase.

Problem Statement

Despite the advancements in AI-driven software development, challenges still persist. One significant issue is accurately assessing and enhancing source code quality. AI-based code generation tools are trained on vast amounts of human-written source code—mined from both public and private repositories—potentially incorporating common vulnerabilities into the generated code. Beyond vulnerabilities, the generated code may also contain code smells, impacting the maintainability of software projects.

Research Questions and Planned Contributions

The primary contribution of this work lies in the experimental assessment of various AI-based tools used for code generation. This objective is formally defined as the primary goal in Section 5.2. The primary goal is further divided into eight research questions, each providing insights into how the assessed AI tools perform in terms of the quality of produced source code. Additionally, there is a secondary goal aimed at deploying an open-source Large Language Model (LLM) and connecting it to an Integrated Development Environment (IDE). This secondary goal contributes a practical approach that could prove beneficial when none of the available AI tools is suitable for a specific use case, such as when complete control over the tools is required. The final type of contribution is an overview of current LLMs and their internal workings.

Method and Approach

The experimental methodology is derived from the Goal Question Metric approach [1]. The primary goal comprises eight research questions, and these questions are addressed using a set of metrics. A total of fifteen metrics are defined for the experiments. Given the substantial number of research questions and metrics, drawing clear conclusions for the primary goal might prove challenging. To address this, a scoring system will be implemented for each research question and the primary goal. During the evaluation of each research question, the tools will accumulate scores based on their performance in the metrics. The tool with the highest score will be selected to draw conclusions for the primary goal. This approach establishes a clear relationship between the research questions and their contributions towards working out the primary goal.

Structure of the Thesis

The first chapter provides theoretical foundations of LLMs along with an overview of state-of-the-art deep learning models commonly employed for source code generation. The second chapter introduces the most popular AI-based source code generation tools and describes their integration with Integrated Development Environments (IDEs). The third chapter delves into the theory of source code quality, with a specific focus on common security vulnerabilities in modern software development. It discusses various methods based on static analysis for detecting vulnerabilities and computing source code quality metrics.

The fourth chapter reviews related works and their findings in the area of AI-generated source code. In the fifth chapter, the data collection process is described, and the goals, research questions, and metrics used in designing experiments are defined. The sixth chapter is concerned with an evaluation of the experiments. Finally, the seventh chapter concludes the work with a summary. The work also includes an appendix providing a description of the attached source code archive and other supplementary material.

1 AI-Driven Code Generation

There are a lot of similarities between natural language and programming languages used in software development. Although natural languages tend to be complex, creative, ambiguous, and pose other characteristics in similar direction, but what humans use from the whole potential set of natural language is more simpler, repetitive and thus more easily modelled with a statistical language model. [2] defines software as *natural* in the same way as natural language, since its created by daily and repetitive human work that shares the same modellable characteristic.

> Programming languages ages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks. [2]

This repetitiveness was studied in [3] and the findings showed that a corpus consisting of 420 million lines of code from over 6000 projects written in C, C++, and Java contained a large degree of repetitions [1]. This finding further supports the argument that source code can be modelled by language models and language models can be in retrospective used in software engineering to aid programmers in their daily work.

Definition 1.0.1 (Language Model [2])**.** Given a set of program lexemes τ, the set of all programs τ^*, and the set of implemented systems S such that $S \subset \tau^*$ a language model is a probability distribution p over individual systems $s \in S$:

$$\forall s \in S, \quad 0 < p(s) < 1 \quad \wedge \quad \sum_{s \in S} p(s) = 1$$

1. The amount of repetitiveness was depending on the size of the code fragment which was compared to the rest of the project as smaller sizes tended to be naturally more repetitive than bigger sizes.

The particular N-gram language model used in [2] is a rather simple statistical model but the authors were still able to show the potential of applying NLP in software engineering.

Definition 1.0.2 (N-gram Model [2])**.** Given that an implemented system s is composed of n tokens a, i.e. $s = a_1 a_2 \dots a_n$ an N-gram model estimates which token has the highest probability of following all the previous tokens. This definition can be expressed in terms of a product of a sequence of conditional probabilities:

$$p(s) = p(a_1) \cdot p(a_2|a_1) \cdot p(a_3|a_1 a_2) \dots \cdot p(a_n|a_1 \dots a_{n-1})$$

In practice, N-gram models leverage Markov property focusing on the previous $N-1$ tokens, i.e. local context to generate the next token.

Definition 1.0.3 (Markov Property [2])**.** Given a 4-gram model, the probability is equal to

$$p(a_i|a_1 \dots a_{i-1}) \simeq p(a_i|a_{i-3} a_{i-2} a_{i-1})$$

Since N-gram models are non-parametric, i.e. values of their parameters [2], are estimated from training data. The usual method of parameter estimation is maximum-likelihood [3]. Authors of [2] used a logical definition of maximum-likelihood function for a task built around repetitions in the training data and that is frequency counting.

Definition 1.0.4 (Maximum-likelihood Based Frequency-counting [2])**.** Given a 4-gram and a wildcard token $*$

$$p(a_4|a_1 a_2 a_3) = \frac{count(a_1 a_2 a_3 a_4)}{count(a_1 a_2 a_3 *)}$$

The last definition is left for a performance metric which in the case of language modelling and distribution estimation on text corpora is cross entropy, which in NLP measures the perplexity of a model when processing a new system.

---

2. or values of parameters of the distributions they model
3. Maximum-likelihood is a general process of finding such parameters of a model or a distribution on a training set, which maximize the probability that the training set was generated from a distribution with the estimated parameters

Definition 1.0.5 (Cross Entropy [2]). Given an implemented system $s = a_1 \dots a_n$ of length n, a language model M, and the probability of the system estimated by the model $p_M(s)$, cross entropy $H_M(s)$ is defined as:

$$H_M(s) = -\frac{1}{n} \log p_M(a_1 \dots a_n)$$

or

$$H_M(s) = -\frac{1}{n} \sum_{i=1}^{n} \log p_M(a_i | a_1 \dots a_{i-1})$$

The basic task of language modelling is then to achieve the lowest value of cross entropy possible. For this purpose a cross validation on the training corpus is a common technique to benchmark performance of statistical or machine learning models.

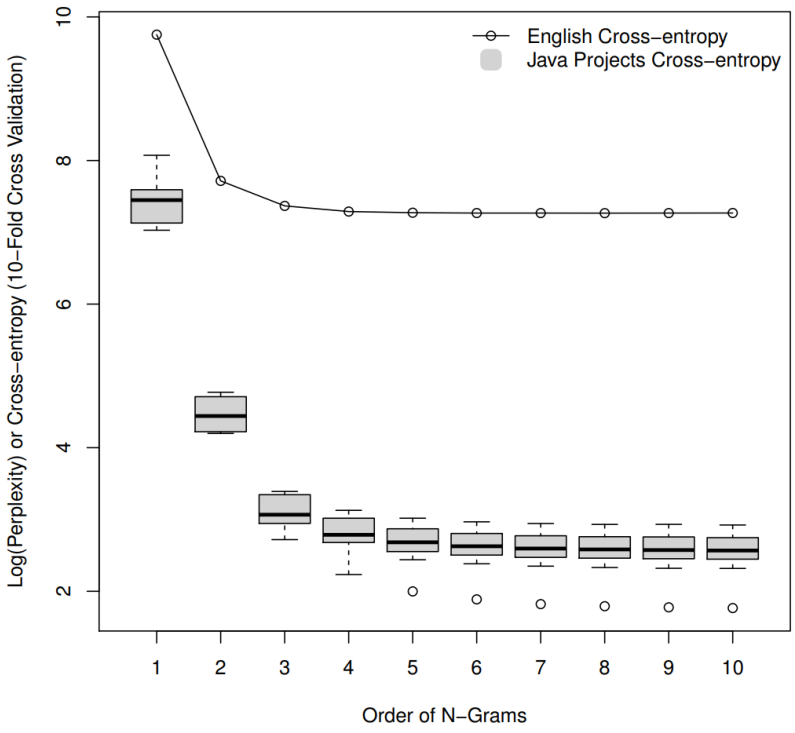


Figure 1.1: The relationship between order of N-Grams and the value of 10-Fold Cross Validation Cross Entropy. Line represents natural language corpus and boxplots represent code corpora. [2]

Figure 1.1 shows that language models performed much better on the code corpus than the English corpus. This leads to the following conclusion which is pivotal for AI-based tools used in software development.

> Corpus-based statistical language models can capture a high level of local regularity in software, even more so than in English. [2]

One may object that the syntax of "artificial" programming languages is more simpler and regular than natural languages and thus the lower values of cross entropy.

> The language models are capturing a significant level of local regularity that is not an artifact of the programming language syntax, but rather arising from “naturalness” or repetitiveness specific to each project. [2]

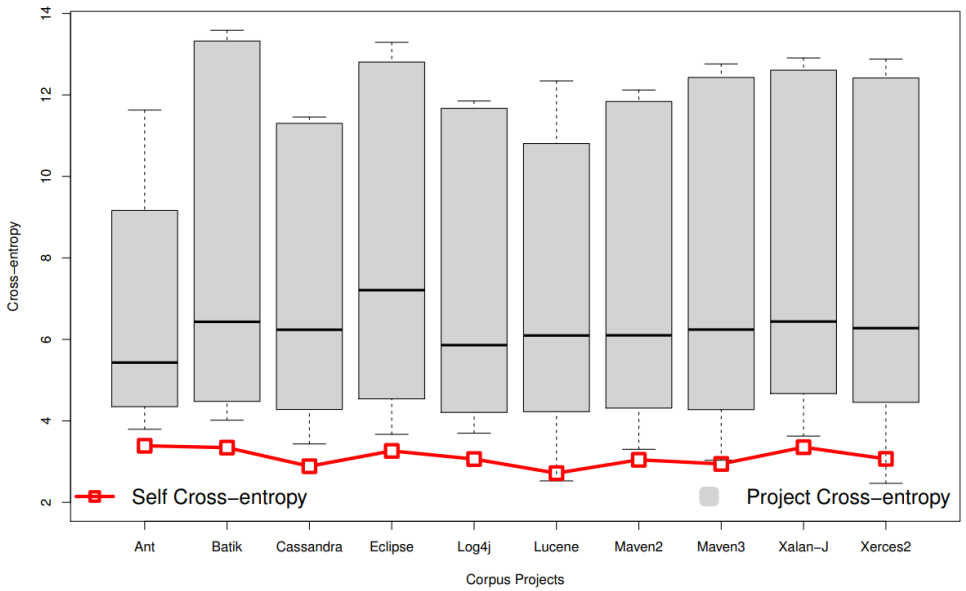


Figure 1.2: The comparison of cross-entropy computed on the training set and cross-entropy computed on other projects. [2]

The results of language modelling in [2] laid foundations for the use of statistical and machine learning NLP methods in software development.

1.1 NLP Applications for Source Code Tasks

The rise of machine learning-driven NLP applications such as machine translation, question answering, text summarization, and language generation has significantly influenced code generation, transforming it into an exclusively machine learning-centric NLP task. From the field of machine learning, the most prevalent and powerful models used in NLP are neural networks, also called deep learning models.

The first type of neural networks used in NLP were Recurrent Neural Networks (RNNs), which were able to process sequences of tokens but suffered from many technical difficulties. The most frequent problem with RNNs was the vanishing gradient problem, which occurred during training of RNNs and was caused by a long series of multiplications of small floating point values of gradients[4] which caused the network to train very slowly or the training even halted at some point and could not proceed further.

The next evolution of neural networks for NLP applications arrived with Long Short-Term Memory (LSTM) networks. LSTMs mitigated the vanishing gradient problem with introduction of gates, which are specialized units responsible for regulating the flow of information within the network and outperformed RNNs in NLP tasks. The current state-of-the-art neural networks for NLP tasks is the transformer neural architecture which will be described in the next section.

Since natural language and source code share similar properties, as was shown at the start of the chapter, advances in standard NLP applications can be transferred to code generation. This means that similar neural model architectures, training methods, datasets, knowledge transfer, and more used in standard NLP can also be used for building code generation neural networks.

4. Gradients are values computed using the back-propagation algorithm based on a defined loss function and a training dataset. Gradients are then used in a training algorithm, such as gradient descent, to adjust the weights of a neural network.

In the current AI-based code generation tools, natural language is used as a bridge to generate code. The transformation of natural language, for instance a description of functionality, to source code is not the only practically used NLP-based task. Numerous applications of NLP deep learning models for source code tasks exist, and they can be categorized based on the type of input that NLP models process.

Code Generation

The most general and applicable NLP task in source code applications. Code generation is an an automated process of transforming natural language specifications or descriptions into executable source code, bridging the gap between human language and programming constructs. Natural language specifications can be in the form of code-level comments, prompts, documentation and other.

Code Completion

A common feature of integrated development environments (IDEs) that suggests and automates the insertion of code elements as developers type, streamlining the coding process and making the developer type less. There have been many types of code completion systems based on rules, type inference, statistical models [5], machine learning models.

Code Suggestion

A subtask of code generation providing developers with intelligent recommendations of code snippets for code enhancements, optimizations, or alternative implementations during the coding phase. This task takes instructions in forms of prompts, code-level comments, function signatures and other.

Code Translation

Also called transpilation, code translation is the conversion of code from one programming language to equivalent code in another pro-

5. N-grams.

gramming language. Code translation facilitates interoperability and adaptation across diverse programming languages, technological environments, and domains. Code transpilation can also be done without the use of NLP or AI-driven approach by using a transpiler.

Code Refinement

Improving or optimizing automatically generated source code. In an NLP setting, code refinement involves enhancing the quality, readability, and efficiency of code that has been generated from natural language specifications or descriptions.

Code Summarization

Creating concise and informative summaries of code snippets or entire codebases to facilitate comprehension, documentation, and knowledge transfer. This task is especially useful for legacy codebases where the human knowledge had been lost or for more exotic languages or technologies which not many software engineers have experience with.

Defect Detection

The identification and analysis of bugs, errors, or imperfections in software code to improve its correctness, reliability, and functionality. Unlike the rests of the task, defect detection can be implemented as a binary classification task, where the input code snippet is categorized either as defective or correct.

Code Repair

Automatic or semi-automatic techniques for identifying and fixing issues or errors in source code. This task provides additional functionality on top of defect detection, as the task implementation must not only recognize that input code snippet contains an error but it must also be able to fix it[6].

6. There is a new field of research called self-healing applications. In this field, applications utilize generative AI to rewrite themselves whenever they detect an error,

Clone Detection

The process of identifying redundant or similar sections of code within a software project to enhance maintainability and reduce redundancy. Clone detection can be seen as a practical application of the DRY[7] software engineering principle.

Documentation Translation

The translation of software documentation from one language to another, ensuring accessibility and comprehension for a broader audience. This task is probably the closest to common NLP tasks such as machine translation since it deals mostly with natural language[8].

NL Code Search

The ability to search for relevant code snippets using natural language queries, enabling developers to locate code based on contextual descriptions rather than on more trivial search methods such as full-text search in the source code repository.

Table 1.1: Summary of code generation NLP tasks.

Task Category	Task
Code-Code	Clone Detection Defect Detection Code Completion Code Suggestion Code Repair Code Translation
Text-Code	NL Code Search Text-to-Code Generation
Code-Text	Code Summarization
Text-Text	Documentation Translation

exception, or a bug and thus can potentially automate bug fixing and debugging. There are several open-source projects which deal with this field.

7. Don't Repeat Yourself - it aims at reducing source code clones and repetitions.
8. But software documentation can of course also contain code snippets.

1.2 Transformer Neural Architecture

The Transformer architecture is a neural network model (Transformers) introduced in [4]. It has since become the foundation for many state-of-the-art NLP models[9], such as BERT or GPT. Transfomers are particularly known for their ability to handle sequential data efficiently and the ability of catching long-range and complex relationships through self-attention mechanisms.

The original Transformer architecture for machine translation was composed of an encoder and a decoder part. The encoder accepts a sequence of input tokens and generates hidden state which is passed to the decoder. The decoder generates sequence of tokens from the hidden state until a special end of output token is generated.

In more detail, the encoder part of a transformer has the following architecture as well as data flow graph:

1. **Tokenization of Input**. The input text is split into a sequence of tokens using a tokenizer. Tokenizer then replaces individual tokens with unique identifiers, usually by using a predefined vocabulary. The tokenizer itself is usually not directly a part of the model. The whole sequence of tokens is fed to the transformer at once[10], by composing them to a matrix or a tensor.

2. **Input Embedding**. Tokens are then embedded into a high-dimensional vector space. Token embeddings contain semantic information about original tokens, i.e. each embedding carries a meaning of the original token. Two similar words are "closer" to each other in the vector space than two unfamiliar words.

3. **Positional Encoding**: Since Transformers do not inherently have a sense of order in their input, as the whole sequence is processed at once, positional encoding is added to the input embeddings to provide information about the positions of tokens in the sequence. Positional encodings can be either absolute or relative and are typically calculated using sinusoidal functions.

9. Transformers are not limited only to NLP but they have achieved state-of-the-art performance also in other areas like computer vision.
10. Unlike RNNs which accept one by one token. This also makes Transformers more computationally efficient than RNNs as they can better utilize modern GPUs.

4. **Multi-Head Self-Attention**. This is the essential building block of the Transformer architecture. The self-attention mechanism computes a series of matrix operations between all elements for each token in the sequence, i.e. the self-attention is *bidirectional* and each token attends to all other tokens in the sequence including itself, which allows the self-attention mechanism capture contextual and relationship information between tokens. Multiple attention heads allow the network to repeat the attention operation with different set of weights and thus allow the network to gather wider spectrum of information from the tokens. Multi-head Self-Attention is described in more detail in the next subsection.

5. **Layer Normalization and Residual Connections**: Layer normalization is applied before each sub-layer (multi-head self-attention and feed-forward network) in the Transformer as a regularization technique to stabilize training. Residual connections ensure that information from previous layers is preserved, enabling gradients to flow easily during training, help prevent overfitting, and contribute to increased performance.

6. **Feed-Forward Neural Networks**: After the attention layers, the Transformer includes Feed-Forward Neural Networks (FNNs). These networks, composed of basic neurons with activation functions, enable the model to process[11] information gathered by the attention layers and generate a hidden state.

Multi-head self-attention, layer normalization and residual connections, and feed-forward neural network form an attention layer. An encoder is composed of 1 to n attention layers. The number of stacked layers in encoder controls the strength of the encoder. Too many attention layers can cause overfitting on too simple tasks or small datasets.

For some NLP tasks, such as language modelling, having an encoder-only Transformer model[12] is enough as feed-forward neural networks are able to generate probabilities from the hidden state using a softmax function.

11. In other words, FNNs introduce non-linearity to the Transformer through the non-linear activation functions of their neurons and multiple hidden layers.
12. A well-known example is the BERT transformer.

The decoder part of Transformers shares similar components with the encoder but exhibits several differences. Notably, the self-attention is *unidirectional*, meaning each token can only attend to tokens on its left. The output layer, equipped with a softmax function, calculates probabilities over the vocabulary to determine the next token, selecting the one with the highest probability. Finally, there is a distinction in the operation of the decoder between training and using a trained decoder for inference.

During training of the original Transformer model for machine translation, decoder takes the hidden state generated by encoder and combines it with the already translated ground truth sentence from training dataset. When generating the *nth* word of the translated sentence, these two inputs are passed into *Masked* Multi-head Self-attention block which masks all words of the ground truth sentence at positions $>= n$. This way, the decoder considers only the relevant part of the ground truth sentence. When the transformer has been trained and is used in inference mode, there are naturally no translated ground truth sentences available. The decoder switches to autoregressive mode and instead of the relevant part of the ground truth sentence it uses the previously generated and translated words from itself.

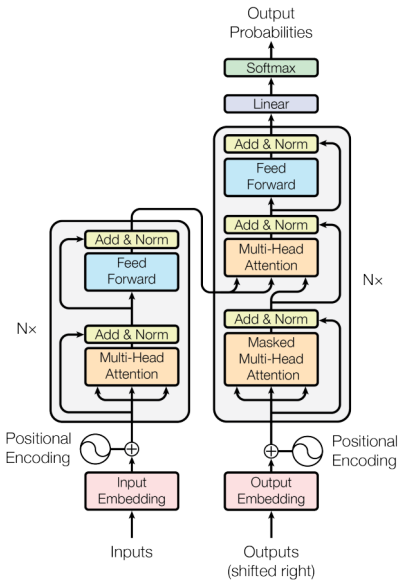


Figure 1.3: Architecture of a Transformer neural network [4].

Attention Mechanism in Transformers

Attention is in the core of the Transformer architecture. It allows transformers to capture contextual and long-distance relationship information between tokens of the input sequence. While there are many versions of attention functions, the dot product and scaled dot product functions are common. The original definition of attention works with vectors which would correspond to a single token in the input sequence, but since Transformers process the whole sequence in a single pass through the network, vectors are stacked into matrices and attention is computed with matrices instead of vectors.

Definition 1.2.1 (Query, Key, Value [4])**.** Let X be the output of the previous network layer, d_{model} the dimension of layers' outputs, d_k the dimension of queries and keys, d_v dimensions of values, and $W^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W^K \in \mathbb{R}^{d_{model} \times d_k}$, $W^V \in \mathbb{R}^{d_{model} \times d_v}$ be the learned projection (weight) matrices of a single self-attention layer. Query Q, key K, and value V are defined as

$$Q = X \cdot W^Q$$
$$K = X \cdot W^K$$
$$V = X \cdot W^V$$

The terms query, key, and value in the definition refer to self-attention instead of general attention. This distinction arises when the query, key, and value are computed from the same input, leading to the use of the term self-attention in such cases. The concept of query, key, and value is based on the logic from information retrieval:

Query. The query represents what information the model is interested in or what it wants to retrieve from the input sequence or the outputs of the previous layer.

Key. The key represents the information contained in the retrievable element (metadata) and how it can be used to provide context for the query.

Value. The value represents the actual information that can be retrieved based on query and key.

The attention mechanism operates by comparing the query with the keys, yielding attention scores or weights. These scores are then normalized to a value of 1 using a softmax function, indicating the degree to which each element in the input sequence — specifically in the value vector — should contribute to the output.

Definition 1.2.2 (Scaled Dot-Product Attention [4])**.** The attention function A is defined as

$$A(Q, K, V) = softmax(\frac{Q \cdot K^\intercal}{\sqrt{d_k}}) \cdot V$$

Compared to standard weights of Feed-Forward Neural Networks which are statically fixed during inference, attention layers compute weights with respect to a particular input sentence and thus are able to dynamically react to different inputs. In layman terms, training of Transformers does not consist of learning a particular set of weights which will be used to process inputs but rather in learning how to estimate what is important based on the particular input sequence. In this view, Transformers are much more flexible than standard Feed-Forward Neural Networks that use a set of fixed weights for all inputs.

Definition 1.2.3 (Multi-Head Attention [4])**.** Let h be the number of attention heads, $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ the projection matrix of the multi-head attention layer. The multi-head attention function MHA is defined as

$$MHA(X) = Concat_{i \in [1,h]}(A(Q_i, K_i, V_i)) \cdot W^O$$

where

$$\begin{aligned} Q_i &= X \cdot W_i^Q \\ K_i &= X \cdot W_i^K \\ V_i &= X \cdot W_i^V \end{aligned}$$

The results from multi-head attention are concatenated and linearly projected using learned linear projections during training. This process allows Transformers to efficiently combine information from multiple attention heads.

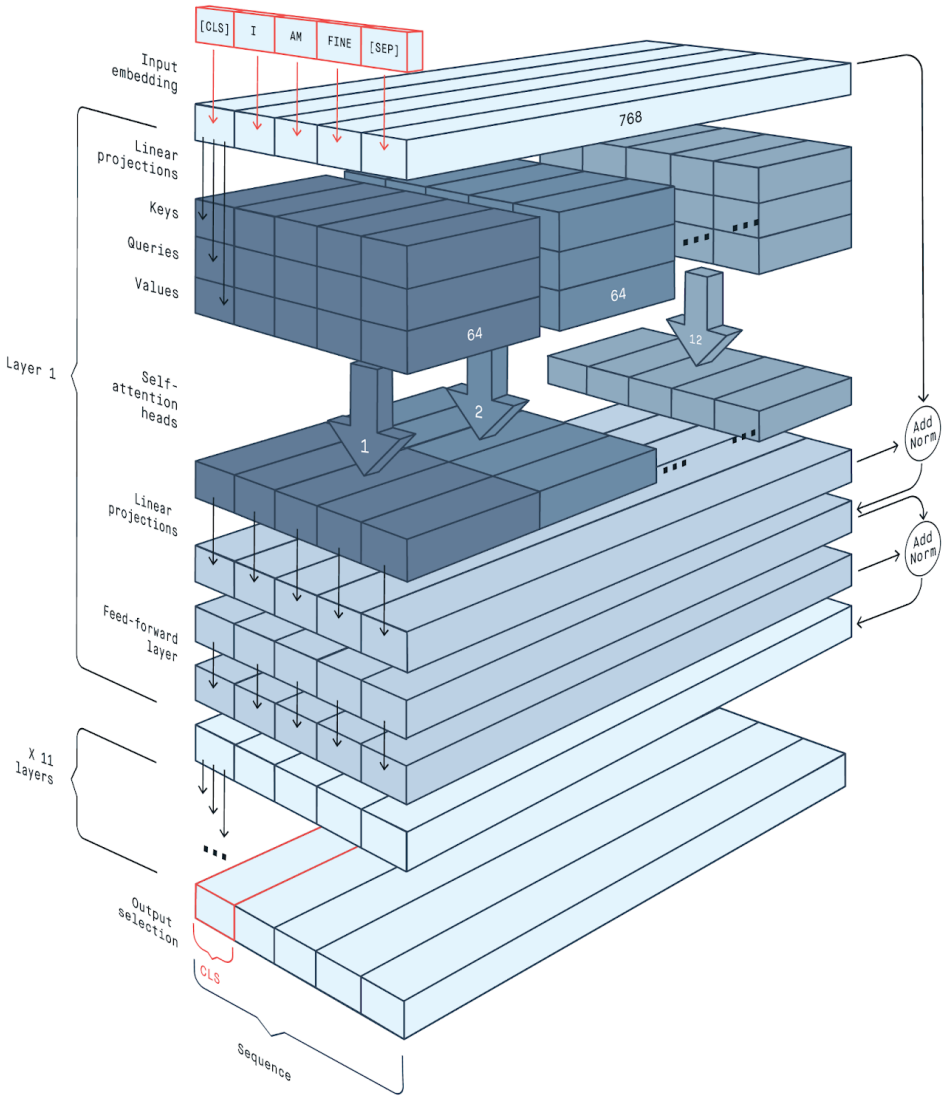


Figure 1.4: Depiction of several parallel attention heads in a single attention block. Queries, keys, and values in different attention heads are differently colored. This figure depicts the capability of the multi-head attention to extract different types of information from the input and then aggregate the obtained information. Source.

1.3 Large Language Models

Large Language Models (LLMs) are the current state-of-the-art types of deep learning models with applicability to wide range of tasks. When looking at the evolution of statistical models for NLP tasks, we can distinguish four development phases. The first phase were n-gram models described at the beginning of the chapter. The second phase included models like Word2vec that used word embeddings and neural networks for NLP tasks. The third phase introduced the Transformer architecture using unsupervised pre-training on larger text corpora and the ability of transfer knowledge from pre-training objectives to downstream tasks. The fourth and the current phase are LLMs which show strong performance and generalization on a wide range of complex tasks.

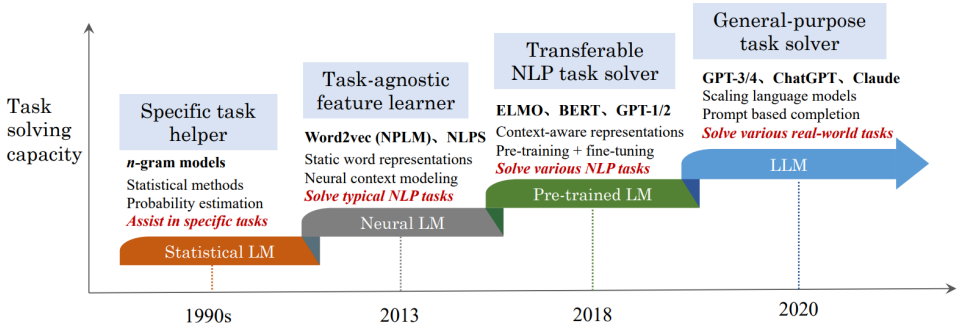


Figure 1.5: The evolution of NLP models considering their task solving capacity [5].

The state-of-the-art performance achievements of LLMs are attributed to their ability to scale effectively with the complexity of models (number of parameters), the size of the training data, and computational resources. It is common for leading LLMs to incorporate hundreds of billions of parameters, undergo training on trillions of text tokens, and utilize hundreds of GPUs for the training process[13].

13. For instance, the LLaMA 2 LLM from Meta comprises 70B parameters. It was trained on 2T tokens and utilized 2000 Nvidia A100 80GB GPUs [6]. Although the training time is not disclosed in the LLaMA 2 paper, its predecessor of a comparable size trained on only 1.4T tokens took 21 days to train on the same 2048 GPUs [7].

The scaling effect of LLMs has been formally expressed as a power-law relationship based on experiments detailed in [8] as

$$
\begin{aligned}
L(N) &= (\frac{N_c}{N})^{\alpha_N}, \quad \alpha_N \sim 0.076, N_c \sim 8.8 \times 10^{13} \\
L(D) &= (\frac{D_c}{D})^{\alpha_D}, \quad \alpha_D \sim 0.095, D_c \sim 5.4 \times 10^{13} \\
L(C) &= (\frac{C_c}{C})^{\alpha_C}, \quad \alpha_C \sim 0.050, C_c \sim 3.1 \times 10^{8}
\end{aligned}
$$

where L is the cross-entropy loss of a model, N is the model size, D is the dataset size, C are the training computational resources, and c is the compute budget.

LLM Architectures

So far, LLMs have been built exclusively on the Transformer architecture, utilizing either both encoder and decoder components or only the decoder. The original architecture, described in Section 1.2, is the encoder-decoder. The decoder-only architecture is futher categorized into **causal decoder** and **prefix decoder**.

The causal decoder employs the same masking mechanism and unidirectional self-attention as the decoder component of the encoder-decoder architecture to autoregressively generate output tokens, but it does so without utilizing the hidden state from the encoder component. Taking machine translation as an example, the original sentence (prefix), concluded with a special EOS (end-of-sequence) token, is input into the decoder to generate the initial output token. Subsequently, each newly generated token is appended after the initial EOS token to the extended sequence, and this extended sequence is input into the decoder again. This process continues iteratively until another EOS token is generated. The sequence of tokens between the initial EOS token and the final EOS token is then extracted as the translated sentence.

Step 0: Prefix + EOS_1
Step 1: Prefix + EOS_1 + Gen. $Token_1$
Step 2: Prefix + EOS_1 + Gen. $Token_1$ + Gen. $Token_2$
Step n: Prefix + EOS_1 + Gen. $Token_1$ + ... + Gen. $Token_n$ + EOS_2

Figure 1.6: An example of a causal decoder autoregressively translating a sentence. Prefix is the original sentence. The sequence $Gen.Token_1 + \cdots + Gen.Token_n$ is the translated sentence.

The causal decoder has an advantage over the encoder-decoder architecture due to a lower number of parameters, utilizing only the decoder component. It is the most used LLM architecture among the leading LLMs [5].

The prefix decoder adapts the unidirectional self-attention of the causal decoder to bidirectional self-attention over the prefix sequence and unidirectional self-attention over the generated sequence. This adaptation allows the prefix decoder to simulate the self-attention of the encoder-decoder architecture without increasing the number of parameters by using an encoder.

Other architectural differences[14] in LLMs include the choice of a specific attention function, primarily aiming to reduce inference latency due to the quadratic complexity of the original attention mechanism. *Full attention* is the attention used in the original Transformer architecture [4]. *Sparse attention* ensures that tokens attend only to a subsequence of the input sequence to reduce the computational complexity of full attention. *Multi-query attention* involves sharing the key and value linear projection matrices among attention heads in a single multi-head attention layer, while the query linear projection matrices remain unique. *Grouped-query attention* defines groups of attention heads that share the same linear projection matrices. *FlashAttention* optimizes the memory layout of attention inputs to utilize faster types of GPU memory. *PagedAttention* improves the memory efficiency of GPUs using paging techniques.

14. More detailed configurations of LLMs include activation functions, absolute/relative position encodings, learning rates, normalization layers, and more.

Pre-Training of LLMs

The success of LLMs relies on their ability to undergo pre-training on extensive text corpora, forming the foundation for their state-of-the-art performance. Typically, the performance of these models is directly proportional to their size, necessitating substantial training data. Given that LLMs often contain billions or even trillions of parameters, a vast amount of training data is crucial.

Consequently, LLMs are trained on datasets consisting of trillions of tokens. Labeling such datasets for supervised training is infeasible for human labelers. However, LLMs employ a specific technique for pre-training on large text corpora known as unsupervised learning. This approach does not require explicit labels in the training dataset; instead, it automatically generates labels based on inherent properties or characteristics of the training data.

These text corpora serve as a valuable resource for creating NLP tasks in an unsupervised manner. LLMs can learn linguistic patterns, structures, language understanding, grammar, reasoning, and other generally applicable skills. Language modeling stands out as one such task where LLMs aim to generate the next word or sequence of words following a given text. The ground truth labels for this task are easily obtained by using the original piece of text from the training corpora.

Another pre-training objective involves filling in masked words within a piece of text and comparing the LLM-generated words to the ground truth words from the text.

Table 1.2: Commonly used text corpora for pre-training of LLMs [5].

Corpora	Size	Source	Description
BookCorpus	5GB	Books	11,000 books
Gutenberg	-	Books	70,000 books
BigQuery	-	Code	code snippets from public repositories
CC-Stories-R	31GB	CommonCrawl	narrative-like style
CC-NEWS	78GB	CommonCrawl	news articles from news sites
REALNEWs	120GB	CommonCrawl	news articles from 5,000 domains
C4	800GB	CommonCrawl	subset of Common Crawl
the Pile	800GB	Other	diverse sources
ROOTS	1.6TB	Other	diverse smaller datasets
OpenWebText	38GB	Reddit links	highly upvoted Reddit links
Pushift.io	2TB	Reddit links	regularly updated entire Reddit content history
Wikipedia	21GB	Wikipedia	Wikipedia articles

Training LLMs poses significant challenges due to the sizes of pre-training text corpora and the vast number of trainable parameters. This process is exceptionally slow, difficult, and resource-intensive in terms of computational resources. Consequently, leveraging suitable parallelization techniques with multiple GPUs becomes inevitable for training[15]. Mainstream techniques involve distributing the training data and the model parameters among GPUs.

Data parallelism involves copying the model to each GPU, where each training batch is divided among the GPUs. The GPUs compute gradients for their respective batches, and these gradients are then aggregated and used in the weight optimization algorithm.

Pipeline parallelism distributes the model's layers among multiple GPUs, allowing parallel computation while minimizing communication overhead caused by layer dependencies.

Tensor parallelism decomposes weight matrices by columns among multiple GPUs, allowing for parallel computation of matrix multiplication on the GPUs. The results are then combined back into matrices[16].

The *ZeRO* technique optimizes data parallelism by avoiding the complete duplication of all model parameters and data across all GPUs. Instead, it distributes fractions of the model parameters and data among the GPUs. The GPUs retrieve the missing parameters or data from other GPUs in an ad-hoc manner.

Mixed precision training utilizes less precise floating-point numbers[17] to decrease GPU memory usage and increase training throughput, potentially at the cost of model performance.

Fine-Tuning of LLMs

The second phase of training LLMs is fine-tuning. Utilizing suitable training procedures can further develop the performance, generalization abilities, applicability to more diverse set of tasks, and stability of predictions of LLMs. *Instruction tuning*, an almost-supervised fine-tuning objective, involves providing LLMs with general training

15. Even for inference, which is orders of magnitude less expensive than training, GPUs are necessary for larger LLMs to achieve reasonable latency.
16. The combination of data parallelism, pipeline parallelism, and tensor parallelism is collectively referred to as *3D parallelism* [5].
17. For instace from 32-bit to 16-bit floating-point numbers.

instruction instances in natural language. It has demonstrated strong generalization performance for unseen tasks and the ability to follow user-specified instructions [5].

Each training instance comprises a task description (instruction), optional input, output, and optional demonstrations. Training instruction instances can be constructed completely from scratch by human labelers or semi-automatically using existing NLP datasets. Taking a question answering dataset as an example, similar questions can be grouped together to create instruction instances with a list of demonstrations, input, and output. A human labeler then adds task descriptions for each instruction instance[18], and the LLM is fine-tuned with this newly crafted dataset. Reusing the original question answering dataset significantly reduces human workload.

The *Alignment tuning* fine-tuning objective aims to align the generated outputs from LLMs with human preferences and values. This process heavily relies on high-quality human feedback. The typical procedure for human feedback generation involves letting the LLM generate a list of answers to queries and having a human labeler rank them according to either objective preferences or a set of rules. This ranking-based system teaches the model to preferentially generate certain outputs over others.

A widely used technique to align the outputs of LLMs with human preferences is *Reinforcement Learning from Human Feedback* (*RLHF*). This reinforcement learning system utilizes a reward model trained on human-ranked outputs of LLM to incentivize and reward the LLM for its output. In inference mode, the reward model predicts human rankings of unseen LLM outputs, and based on the ranking score, computes the LLM's reward. The reward can be positive or negative, with the latter encouraging the model to generate more reliable and safer outputs according to human preferences[19].

---

18. The entire question-answering dataset could be augmented with a single human-written task description (e.g., 'Please answer this question' [5]). Nevertheless, the effectiveness of employing such a general task description for all examples in the dataset is questionable.

19. The use of the RLHF algorithm in alignment tuning is known to push safe behavior to an extreme, often leading LLMs to generate outputs that are deemed safe but not really useful. This challenge is recognized as the evasion problem [5].

Generating Outputs From LLMs

A fully trained LLM can be employed for a variety of tasks. As outlined in Section 1.2, deep learning models utilizing the Transformer architecture produce a vector of probabilities across a predefined vocabulary. The vocabulary entry with the highest probability is then generated:

$$token = \underset{x \in \text{vocabulary}}{\arg\max} \; P(x|\text{previously generated tokens}).$$

This decoding strategy is referred to as *greedy search*, enabling LLMs to produce satisfactory outputs in tasks where results heavily depend on the inputs. Greedy search can be further enhanced through *Beam search*, which explores multiple tokens at each generation step. At each step, n (based on the beam size parameter) tokens with the highest probabilities are selected, and all n possible sequences are retained. The generation process continues for each possible sequence until they all produce the EOS token. The sequence with the highest joint probability of its tokens is then chosen as the resulting sequence.

In more creative tasks such as content generation, creative writing, summarization, and code generation, where there is not a single correct output, conventional decoding strategies might not be optimal [5]. To adapt LLMs for proficiency in these creative and open-ended tasks, decoding strategies need enhancement by introducing a certain degree of nondeterminism. This is typically achieved through sampling methods, where the next generated token is randomly sampled from a probability distribution:

$$\text{token} \sim P(\text{token}|\text{previously generated tokens}).$$

While the most likely token still carries a high probability of being generated, other tokens with significant probabilities[20] can also be generated. As a result, running an LLM multiple times with the same input sequence can produce diverse outputs.

To avoid sampling tokens with very low probabilities, a *temperature* parameter is introduced in the softmax function, adjusting the

20. For meaningful outputs, the most likely token and the set of the second most likely tokens share lexical semantics (e.g., the most likely token is "dog," and the set of other likely tokens is "cat," "rabbit," "hamster").

probability distribution toward more likely tokens. The lower the temperature parameter, the more deterministic the outputs become, and vice versa. Certain LLM-based tools let users adjust the temperature parameter, providing control over the creativeness of LLMs.

LLM User Interface

Most end users interact with LLMs using a prompt interface. *Prompt* can be defined as an instruction or a query to an LLM to solve a task. The way prompts are written heavily influences the quality of LLM outputs. Low quality prompts, confusing prompts or prompts with unclear task definition will most probably generate undesired outputs.

The majority of end-users interact with LLMs through a prompt interface. A *prompt* can be defined as an instruction or query presented to an LLM to accomplish a task. The quality of *prompt engineering* heavily influences LLM outputs. Low-quality, confusing, or unclearly defined prompts are likely to yield undesired outputs.

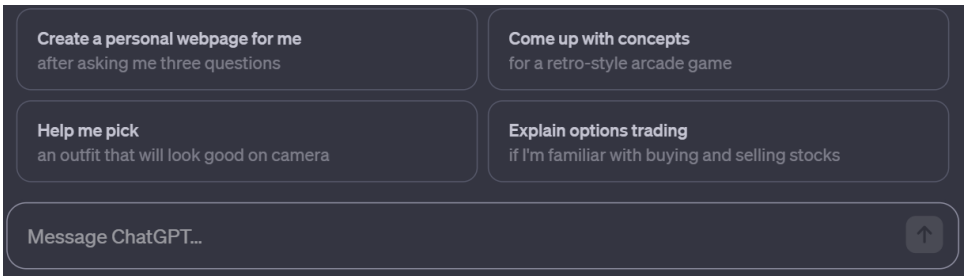


Figure 1.7: Web-based prompting interface of ChatGPT.

[5] identifies four main components of prompts: task description, input data, contextual information, and prompt style. Selecting the appropriate prompt style for a specific domain or task can significantly enhance the quality of responses. In handling complex tasks, it is advisable to decompose the task into sub-tasks, facilitating easier interpretation for LLMs. As LLMs are frequently fine-tuned using instruction-based objectives, improving the quality of responses for specific tasks can be achieved by incorporating a few demonstrations within the prompt eliciting generalization performance on unseen tasks.

1.4 Code Generation Large Language Models

In recent years, LLMs have gained significant attention, entering a hype cycle and giving rise to a vast ecosystem of LLMs distinguished by unique architectures, training data, training methods, and domains. LLMs capable of code generation and related tasks are built on the same principles outlined in the previous section about LLMs[21], differing mostly in their choice of code-oriented datasets for fine-tuning. In this section, we present representatives of leading open-source and closed-source state-of-the-art LLMs capable of code generation and related tasks.

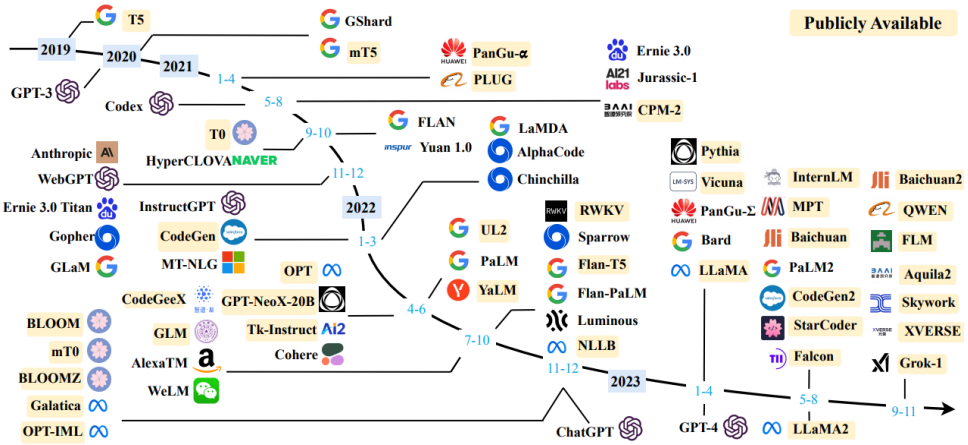


Figure 1.8: The landscape of LLMs [5].

Generative Pre-trained Transformers

The Generative Pre-trained Transformer (GPT) model family is arguably the most well-known among LLMs. Developed by OpenAI, these decoder-only models have played a pivotal role in shaping the evolution of LLMs with each new generation. Currently, there are four major foundational models: GPT-1, GPT-2, GPT-3, and GPT-4, with GPT-5 announced but not yet released. The first two generations have become obsolete and have been open-sourced. GPT-3 and GPT-4

21. In fact, the same LLM can be used for both machine translation and code-related tasks, given that suitable fine-tuning objective were employed during its fine-tuning.

have both seen upgrades in the form of new minor versions, namely GPT-3.5 Turbo and GPT-4 Turbo. However, precise technical details about the architecture, training methods, and training data for both GPT-3 and GPT-4, as well as their newer minor versions, have not been disclosed in their official publication papers [9, 10].

GPT-1. Released in 2018, GPT-1 was the first model in the GPT series, containing 117M parameters. It combined unsupervised pre-training for language modeling with supervised fine-tuning [11].

GPT-2. Released in 2019, GPT-2 scaled the number of parameters to 1.5B. In contrast to the previous generation, it relied solely on unsupervised training by predicting the next word in a sequence. The authors aimed to unify all NLP tasks as word prediction tasks [12]. However, GPT-2 was found to be insufficiently complex for employing only unsupervised training, as its fine-tuned versions achieved better performance [5].

GPT-3. Released in 2020, this generation marked a significant leap in model size with 175B parameters and can be regarded as the inception of LLMs. It employed in-context learning, enabling it to excel in unseen tasks without requiring changes to the model's parameters. The substantial size of 175B parameters empowered the model to showcase remarkable capabilities for NLP tasks [9].

Codex. Released in 2021, GPT-3 was noted for its limitations in solving complex reasoning problems, including code generation and mathematical reasoning [5]. In response, OpenAI introduced the Codex model, built upon the GPT-3 architecture but with up to 12B parameters. This model was fine-tuned on a multi-lingual code dataset sourced from repositories on GitHub, resulting in improved performance in these specific tasks [13]

GPT-4. Released in 2023, GPT-4 continued the trend of increasing model size and complexity, achieving an impressive 1.76T parameters, making it one of the largest models ever created. Its newer minor version, GPT-4 Turbo, introduced task capabilities beyond the scope of NLP, encompassing text-to-image generation, text-to-speech generation, and computer vision [10].

LLaMA

LLaMA, a relatively new family of open-source LLMs from Meta, has become a center of attention in LLM research. These models serve as the foundation for many researchers, given their open-source nature and the authors' release of extensive evaluations on numerous benchmarks. LLaMA models have found applications as base models for more specialized variants, frequently undergoing fine-tuning with instruction tuning objectives tailored to specific domains [5].

LLaMA 1. Released in 2023, LLaMMA 1 is available in versions with 6.7B, 13B, 32.5B, and 65.2B parameters. The training procedure employed a combination of common open-source datasets, including CommonCrawl, C4, GitHub, Wikipedia, Books, ArXiv, and StackExchange, totaling 1.4T tokens. The authors' emphasis on the openness of LLM research is evident in their choice of datasets. Notably, LLaMA 1 with 13B parameters has demonstrated superior performance compared to GPT models from OpenAI, including GPT-3 with 175B parameters, across most open-source benchmarks [7].

LLaMA 2. Released in 2023, the second generation of LLaMA models was trained on an even larger dataset, sourcing data from publicly available repositories, totaling 2T tokens. LLaMA 2 introduces a general foundational model and a specialized version, LLaMA 2-CHAT, specifically optimized for dialogue. Both variants are available in versions with 7B, 13B, 34B, and 70B parameters [6].

Code LLaMA. Released in 2023, Code LLaMA is a family of open-source LLMs based on LLaMA 2, sharing a foundation pre-trained on 2T tokens. This family underwent additional training on code tokens, with the foundational model, Code LLaMA, trained on 500B tokens, Code LLaMA - Python on 500B tokens and 100B Python tokens, and Code LLaMA - Instruct on 14,000 instruction examples. All versions are available in 7B, 13B, and 34B parameter models [14].

Hugging Face

Hugging Face is a company that maintains the largest open-source curated library of transformer neural networks, containing over 400,000 models [15]. Naturally, they also design and train their own open-source LLMs, which they make available as part of the library.

CodeParrot. Released in 2021, CodeParrot is built on a similar architecture as GPT-2, featuring models with 110M and 1.5B parameters. The training dataset comprised 30B tokens. Due to the comparatively smaller size of both the models and training data in comparison to other code-oriented LLMs, CodeParrot outperformed only smaller models [16].

StarCoder. Released in 2023, the base model StarCoderBase, featuring 15.5B parameters, was trained on 86 programming languages and 1T tokens from the open-source The Stack dataset. The fine-tuned version, StarCoder, underwent additional training with 35B Python code tokens [17].

Falcon. Released in 2023, Falcon comes in model versions with 1.3B and 7.5B parameters. This model was trained on an impressive 5T tokens obtained from the CommonCrawl database [18]

DeepMind

DeepMind, a company owned by Google, has developed a series of LLMs with a primary focus on innovating training techniques.

Chinchilla. Released in 2022, Chinchilla is a general LLM with 70B parameters, trained on 1.4T tokens. The primary purpose of this model was to estimate the optimal scaling law of LLMs, and it demonstrated superior performance to GPT-3 on the Massive Multitask Language Understanding benchmark [19].

AlphaCode. Released in 2022, AlphaCode is an LLM available in versions with 300M, 1.1B, 2.8B, 8.7B, and 41.1B parameters. It is specifically designed for competition-level code generation with special requirements [20]. AlphaCode employs a novel training procedure based on reinforcement learning and clustering of suggested programs. As a result of this innovative approach, AlphaCode is a state-of-the-art code generation LLM in the competitive programming domain [20].

Table 1.3: LLMs Summary.

Company	LLM Name	Parameter Versions	Domain	License
OpenAI	GPT-1	117M	general	open
OpenAI	GPT-2	1.5B	general	open
OpenAI	GPT-3	175B	general	closed
OpenAI	GPT-4	1.76T	general	closed
OpenAI	Codex	12M to 12B	code generation	closed
Meta	LLaMA 1	6.7B, 13B, 32.5B, 65.2B	general	open
Meta	LLaMA 2	7B, 13B, 34B, 70B	general	open
Meta	LLaMA 2-CHAT	7B, 13B, 34B, 70B	chatbot	open
Meta	Code LLaMA	7B, 13B, 34B	code generation	open
Meta	Code LLaMA - Python	7B, 13B, 34B	Python code generation	open
Meta	Code LLaMA - Instruct	7B, 13B, 34B	code generation	open
HuggingFace	CodeParrot	110M, 1.5B	code generation	open
HuggingFace	StarCoder	15.5B	code generation	open
HuggingFace	Falcon	1.3B, 7.5B	general	open
DeepMind	Chinchilla	70B	general	closed
DeepMind	AlphaCode	300M, 1B, 3B, 9B, 41B	competitive programming	closed

Risks of LLMs

LLMs are recognized for not being optimal models in terms of risks and security. This is primarily attributed to the fact that LLMs are trained on large and diverse datasets harvested from the public internet, and their content cannot be fully controlled. In the context of LLM-based tools for software development, which are primarily aimed at generating source code rather than natural language, they are not particularly influenced by the relatively common social risks associated with LLMs, such as bias, racism, toxicity, and hate speech—except, perhaps, in the case of chatbot-based or prompt-based code generation LLMs like ChatGPT, which generate a substantial amount of complementary text.

The primary risks associated with code generation LLMs include hallucinations, trustworthiness, and security. Particularly with junior and inexperienced programmers, there is a notable risk of accepting incorrectly generated, vulnerable, or otherwise suboptimal code. Factors such as time pressure, lack of experience, or reluctance to conduct thorough testing contribute to this risk. This work is motivated, in part, by the need to illuminate the risks associated with code generation LLM-based tools and to address questions related to source code quality through the evaluation of a series of experiments.

Hallucinations

In the context of LLMs, hallucinations refer to instances where the model generates content that is factually incorrect, fictional, or entirely fabricated. These occurrences can arise when the model extrapolates information based on patterns learned during training but produces outputs that have no basis in reality. Taking code generation as example, an LLM generates code containing fictional libraries, language syntax, or the LLM tries to convince the user that the generated code is correct even though it is incorrect.

Hallucinations can also arise from to limited knowledge of LLMs. A common method to mitigate hallucinations is to extend the LLM's knowledge base beyond the statically fixed training dataset. This can be achieved by utilizing a vector database together with a suitable information retrieval method, such as *retrieval augmented generation* (*RAG*). Retrieved information is then incorporated into the user's prompt to provide the LLM with additional knowledge and context,

Trustworthiness

Trustworthiness concerns the reliability and dependability of the information generated by LLMs. Trustworthy models should consistently produce accurate and verifiable content, instilling confidence in users regarding the correctness of the generated information. Ensuring the trustworthiness of LLMs is challenging due to potential biases in the training data, the model's vulnerability to adversarial attacks, and the difficulty in verifying the accuracy of generated content, especially in dynamic or rapidly changing contexts, such as software development.

Security

In the context of code generation LLMs, security concerns involve potential vulnerabilities in the training data that could be memorized by the LLM and transferred to the generated code. Another security concern is the potential leakage of private and sensitive data that could also be memorized by the LLM from the training dataset.

2 AI-driven Software Development Tools

This chapter introduces specific LLM-based tools desgined for integration into the software development process, with a primary emphasis on source code generation. The focus will center on complete tools or software components, given their comprehensive features and higher likelihood of adoption by software engineers compared to raw LLMs.

Interfacing

Software engineers commonly use Integrated Development Environments (IDEs) for writing source code. Therefore, code generation tools focus on seamless integration with popular IDEs such as Visual Studio Code, IntelliJ IDEs, Neovim, and others, allowing engineers to work within their familiar environments. The integration of code generation tools into IDEs is achieved through plugins. These plugins link IDE actions, such as key presses and other commands, to requests sent to servers hosting the LLMs or to servers containing logic that, in turn, dispatches the requests to LLMs hosted in other locations such as the cloud or Software as a Service (SaaS) platforms. The implementation of plugins is IDE-specific and relies on the respective IDE's software development kit or protocol. For example, Intellij IDEA plugins are predominantly written in Java, Visual Studio Code plugins in JavaScript, and Neovim plugins in Lua.

Some code generation tools, such as ChatGPT, offer a web-based interface or a web API without official plugins for their integration into IDEs. In such cases, third-party plugins may exist, facilitating their integration with IDEs. Moreover, there are cases where code generation tools come with their own dedicated IDEs.

GitHub Copilot

GitHub Copilot is arguably the most popular AI-based code generation tool, having over a million paying users and being adopted by more than 37,000 organizations [21]. According to a survey conducted by GitHub, developers using Copilot reported completing required tasks in 55% less time [22], showcasing Copilot's potential to enhance software development efficiency.

Copilot is built on the OpenAI Codex LLM and has been actively developed by GitHub since June 2021 [21]. Supporting a wide array of programming languages and integration with numerous IDEs, Copilot provides real-time code suggestions, inline completions, suggests code based on comments in natural language, and features a prompt-style interface known as GitHub Copilot Chat, which utilizes prompting for code generation.

GitHub Copilot is a subscription-based service and is available as part of GitHub's paid plans. These plans include Copilot Individual for individual developers, and Copilot Business and Copilot Enterprise for organizations. The subscription costs are $10, $19, and $39 per user per month, respectively [21].

A notable feature of GitHub Copilot is the completions panel, which enables the synthesis of multiple solutions simultaneously, allowing users to choose the most suitable one. Copilot offers advanced configuration options, including the temperature parameter, which controls the level of creativity in Copilot's outputs. Users can also configure the number of synthesized solutions in the completions panel and view additional information, such as the mean probability of the suggested code snippet. Additional configuration parameters include the maximum number of generated tokens and the maximum number of suggested lines of code.

ChatGPT

ChatGPT is a closed-source general chatbot developed by OpenAI in 2022 [23]. It utilizes multiple GPT LLMs as its backend, offering a free version with GPT-3.5 and a paid plan with GPT-4, priced at $20 per month. For humans accessible through a web interface, ChatGPT is often employed by various tools via an API for diverse tasks. While ChatGPT is not exclusively code-oriented, it is a popular tool for various software development tasks, primarily source code generation. Users access ChatGPT through prompting, and it does not provide default support for IDE integration or real-time code generation within editors.

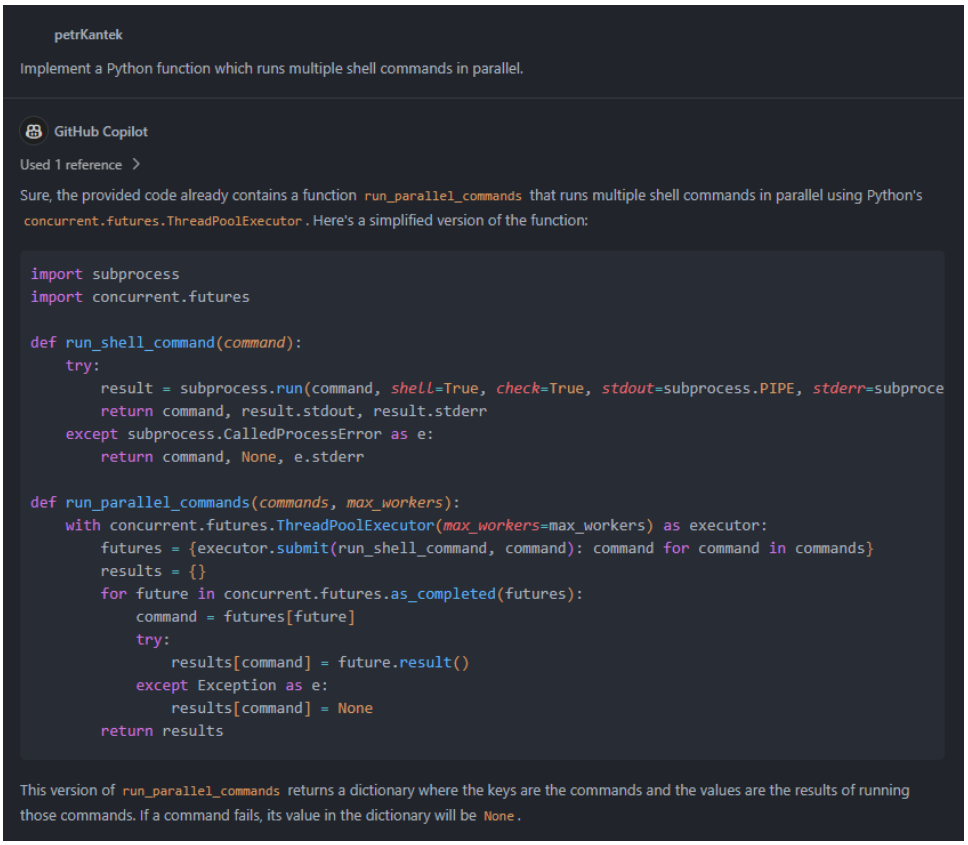


Figure 2.1: GitHub Copilot Chat.

Tabnine

Tabnine is a closed-source tool that has been around for many years. It utilizes a collection of multilingual GPT models, but more detailed information about the architecture or the use of data sets is not public [24]. Tabnine offers features such as whole-line code completions, full-function code completions, and natural language-to-code completions. Users can choose from three plans: Starter (free), Pro ($12 per user per month), and Enterprise (quoted individually). When using the Tabnine Pro plan, the user can choose from local, cloud, or hybrid model serving modes. An advanced feature of Tabnine is its ability to be trained on private code repositories.

CodeGeex

CodeGeeX is an open-source and free tool for source code generation. The newest utilized LLM is multilingual CodeGeeX2 [25]. It offers standard features, programming languages, and integration with common IDEs. Similarly to Tabnine, CodeGeex also offers an Enterprise plan which allows organizations to have the tool fine-tuned on their code repositories.

Other popular AI-based tools for software development include Replit GhostWriter [26], Codeium [27], Blackbox [28], and Cursor [29]. Cursor is somewhat unique in that it provides its own IDE, which has been designed from scratch around AI-driven software development

Custom AI-driven Code Generation Tool

While paid cloud-based services for code generation, such as GitHub Copilot and Tabnine, are suitable for many use cases and fields of application, there may be circumstances when individuals or enterprises would benefit from having their own self-hosted AI-driven tools. This section describes the steps involved in deploying a custom tool. But firstly, it is important to identify the specific reasons that could lead to the decision to have a custom AI tool, along with the associated benefits and risks, include:

1. Missing new libraries
2. Missing languages or domains
3. Enterprise data security and compliance
4. Cost optimization
5. More accurate code suggestions
6. More control over the entire tool and potential further development
7. Fine-tuning on private codebases

Configuring a Pre-Trained LLM

Utilizing an open-source pre-trained LLM on an extensive data set of source code represents the simplest use case. In this scenario, one only needs to download the pre-trained model or its weights and expose it via an HTTP API. As no additional training is required, the cost and implementation time for this use case are the smallest among all cases.

Fine-tuning a Pre-Trained LLM on a Private Codebase

Performing fine-tuning on a pre-trained model using a private codebase involves adapting the model to the specific patterns and characteristics of the proprietary source code. This process enhances the model's performance and relevance to the unique coding conventions within the organization. Custom fine-tuning of an LLM is considerably laborious. The necessary steps can be summarized as:

1. Create a fine-tuning data set
2. Download a pre-trained model
3. Set up training hardware resources
4. Write a training loop
5. Deploy the model
6. Validate the code suggestions.

Creating a fine-tuning data set, setting up sufficient hardware resources for training, and the training process itself are the most time-consuming steps of the whole process.

Deploying Own Model

The model can be deployed in several ways. *On-premise server.* Deploying the model on a standard HTTP-based web service on private server which accepts requests with code context and responds with code suggestions. The responsibility for operating the model lies entirely in the hands of the programmer or engineer.

Cloud hosting. Employing the same principles as an HTTP-based web service mentioned in the previous deployment method, this service is hosted in the cloud. It accepts requests with code context and delivers code suggestions, leveraging the scalability and accessibility advantages of cloud infrastructure.

Local deployment. Involves deploying the model on the same machine as the programmer, allowing for immediate access and operation within the local development environment. This solution is not feasible for bigger LLMs.

Creating Interface for a Deployed Model

The final step is to link the deployed model to an interface tailored for programmers. In the case of *copilot interface*, the approach involves generating real-time, in-line code suggestions as programmers write code within their IDE. Achieving this functionality entails the creation of an IDE plugin, establishing a seamless connection between the deployed model and the programmer's coding environment.

The *chatbot interface* is designed for conversation-style coding, wherein the programmer submits prompts to the chatbot, and the chatbot responds with code suggestions and other complementary information.

3 Source Code Quality

Quality of software is an essential topic of software development. In the current, rather agile, software development life cycle, ensuring the high quality of source code has become a common task throughout the software development life cycle. The assessment can be done whenever a new functionality is added to the code base, which in technical terms translates to assessing the quality of each pull request. The assessment is carried out by experienced developers based on measurable properties, i.e. metrics, of source code or even automatically by comparing source code metrics' values to default thresholds for the project or enterprise-level requirements.

Software quality has been standardized by two international standards, ISO/IEC 9126 [30] and its successor ISO/IEC 25010:2011 [31][1]. These standards have played a pivotal role in shaping understanding and assessment of software quality. ISO/IEC 9126 defines a quality model having functionality, reliability, usability, efficiency, maintainability, and portability as the main characteristics of software quality. The successor, standard ISO/IEC 25010:2011, added security and compatibility to the model. The quality models are primarily associated with non-functional requirements which go beyond particular features of software.

These two standards defined a broad perspective of software quality distinguishing between internal and external quality metrics. This work researches the quality of AI-generated code in terms of internal quality metrics of source code, i.e. static measures. Internal quality metrics will be described in the following two sections. The first section is focused on software security, particularly common vulnerabilities found in source code and how they can be automatically detected using static analysis of source code.

The second section introduces static metrics for measuring the following source code characteristics: readability, complexity, code smells, maintainability, technical debt, consistency, modularity, reusability, testability, and documentation.

---

1. After the standard ISO/IEC 9126 became obsolete, ISO developed a whole family of standards called SQuaRE (Software product Quality Requirements and Evaluation), comprised of standards with identifiers ISO/IEC 250xy.

3.1 Software Vulnerabilities

Software vulnerabilities are weaknesses or flaws in a software system that can be exploited by attackers to compromise the security, integrity, or availability of the system [32]. Vulnerabilities are not necessarily considered as bugs, since they don't affect the functionality of software directly, but an effort from an attacker must be involved in order to exploit the vulnerabilities to compromise software.

Vulnerabilities can exist in various components of a software application, including the code, configuration settings, libraries, or dependencies. Exploiting these vulnerabilities can lead to unauthorized access, data breaches, system crashes, or other security incidents.

There has been significant work done in analysing and documenting vulnerabilities which gave rise to many databases and lists of commonly found vulnerabilities. This effort is often sponsored by governmental cybersecurity agencies as mitigating vulnerabilities in information systems and other applications of critical infrastructure is essential in the 21st century.

OWASP

Open Worldwide Application Security Project (OWASP) is a non-profit organization operating in the field of cyber security and particularly in web application security. OWASP creates and regularly updates **OWASP Top 10** which is a list of the most critical categories of vulnerabilities found in web applications[2].

The list is created by gathering data[3] from companies and from conducting surveys among cyber security professionals. From the selected 10 categories appearing in the final list, 8 categories are based on the collected data and 2 categories are from the surveys. The collected data are analyzed in terms of the incidence rate of around 400[4] unique Common Weakness Enumerations (CWEs) which are mapped to 10 categories.

2. The newest version of OWASP Top 10 is from 2021.
3. The dataset consists of over 500 000 web applications.
4. OWASP Top 10 2021 increased the number of analyzed CWEs from 30 in OWASP Top 2017.

Table 3.1: OWASP Top 10 2021 [33]

Category	Description
A1-Broken Access Control	Inadequate access controls that allow unauthorized users to perform actions or access data
A2-Cryptographic Failures	Exposure of sensitive data due to inadequate protection or weak encryption
A3-Injection	Attacker sends malicious data as part of a command or query to exploit vulnerabilities in data parsing and processing
A4-Insecure Design	Missing or ineffective control design
A5-Security Misconfiguration	Insecurely configured settings, default accounts, and unnecessary services that may expose vulnerabilities
A6-Vulnerable and Outdated Components	Insecurely configured settings, default accounts, and unnecessary services that may expose vulnerabilities
A7-Identification and Authentication Failures	Weaknesses in authentication mechanisms, such as credential stuffing, weak password policies, and insecure session management
A8-Software and Data Integrity Failures	Relates to code and infrastructure that does not protect against integrity violations
A9-Security Logging and Monitoring Failures	Inadequate logging and monitoring, making it difficult to detect and respond to security incidents
A10-Server Side Request Forgery	Injection of malicious URLs from user input and crafted requests

Common Weakness Enumeration

The Common Weakness Enumeration (CWE) is a category system with over 600 categories of common software and hardware weaknesses, design flaws, and programming errors funded by the U.S. Department of Homeland Security and Cybersecurity and Infrastructure Security Agency and managed by The MITRE Corporation. The categories form a hierarchy based on their level of abstraction. CWE also provides several views of the categories: the Software Development view organizes categories suitably for analysis during software development, the Hardware Design view organizes categories suitably for hardware design, and the Research Concepts view supports security research by organizing the categories by their behaviors [32].

Similarly to OWASP, CWE also publishes a list of most critical vulnerabilities in their CWE Top 25 list. The list is based on data from the U.S. National Vulnerability Database (NVD) with over 40000 records. Each record is examined by its root cause and assigned to a CWE category. The categories are then ranked by their danger score and the most dangerous 25 categories are selected for the CWE Top 25 list.

Definition 3.1.1 (Danger score [32]). Let CWE_X be a CWE category X, its danger score is computed as the product of its min/max normalized values of frequency rating Fr and severity rating Sv

$$\text{Score}(CWE_X) = \text{Fr}(CWE_X) \times \text{Sv}(CWE_X) \times 100$$

where

$$\text{Fr}(CWE_X) = \frac{\text{count}(CWE_X \in \text{NVD}) - \min(\text{Freq})}{\max(\text{Freq}) - \min(\text{Freq})}$$

$$\text{Sv}(CWE_X) = \frac{\text{average\_CVSS}(CWE_X) - \min(\text{CVSS})}{\max(\text{CVSS}) - \min(\text{CVSS})}$$

$$\text{Freq} = \{\text{count}(CWE_{X'} \in \text{NVD}) \text{ for each } CWE_{X'} \text{ in NVD}\}$$

Since both OWASP Top 10 and CWE provide good basis for vulnerability analysis, but they are at different levels of abstraction, The MITRE Corporation published an official tree-like mapping between OWASP Top 10 and CWE categories.

Table 3.2: The first 15 entries of CWE Top 25 2023 sorted from most critical vulnerabilities [32].

ID	Description
CWE-787	Out-of-bounds Write
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-416	Use After Free
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-20	Improper Input Validation
CWE-125	Out-of-bounds Read
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-862	Missing Authorization
CWE-476	NULL Pointer Dereference
CWE-287	Improper Authentication
CWE-190	Integer Overflow or Wraparound
CWE-502	Deserialization of Untrusted Data

Vulnerability Analysis

Analysing software code for detection of vulnerabilities is a crucial aspect of software security. Various methods and techniques are used to identify vulnerabilities in software code. The most basic one is manual inspection of source code by security experts, which is necessary in complex cases or when there is a completely new vulnerability discovered but this method does not scale and cannot be automated. Penetration tests detect Vulnerabilities by simulating cyberattacks. The focus of the next section will be on automated static analysis of source code for detecting vulnerabilities using the CodeQL query language.

CodeQL

CodeQL is a declarative, object-oriented logic programming language for querying relational data models [34]. CodeQL is a general-purpose query language with SQL-like syntax and Datalog semantics, but it has found its application mainly in static analysis of source code[5]. It is in the core of the GitHub CodeQL tool which is used for security analysis of source code[6] [34]. GitHub CodeQL allows software engineers and researchers perform detection of vulnerabilities in code bases using a query language as if a code base was a database[7] and vulnerability a piece of information to be retrieved by a query.

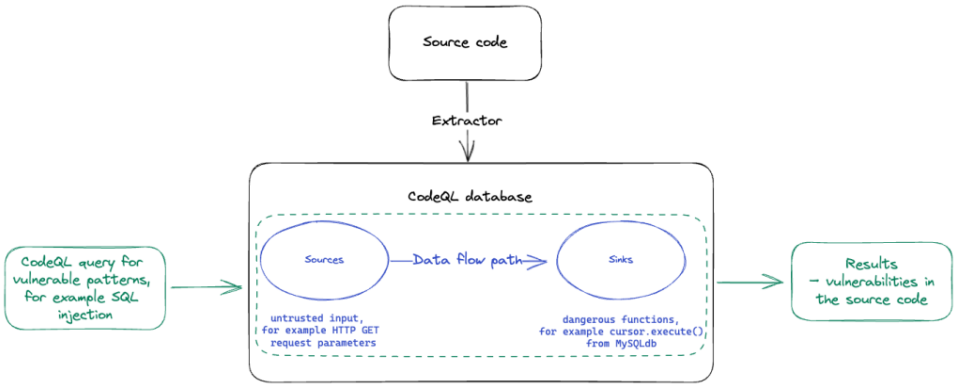


Figure 3.1: CodeQL analysis schema. Source.

GitHub CodeQL uses two main methods of static analysis: data flow analysis and taint analysis. Data flow analysis tracks the flow of data through a program. This is achieved by implementing a query for vulnerable patterns which tracks the data flow between a source and a sink. Source is an untrusted input of a program which is passed to a sink section of code, which executes sensitive operations, for instance

5. The goal of static analysis using CodeQL can encompass various objectives, such as vulnerability analysis, identification of violations of private enterprise-level coding rules, detection of code smells, and more.
6. Originally developed by Semmle, which was then acquired by GitHub.
7. The program's source code is not stored directly in the database, but its representation is created by a language-specific extractor and stored in the database. The representation of source code can be thought of as an abstract syntax tree or a control flow graph.

database reads. This particular data flow can become a security vulnerability causing malicious behavior, such as unauthorized data read. If the source and sink are not securely implemented, the query is able to detect it and the code can be rewritten in a more secure way.

Standard data flow analysis poses a limitation since it is able to track only value-preserving data [34]. To solve this limitation, the tool uses a modified form of data flow analysis called taint analysis that allows tracking of data that does not preserve its value. Throughout taint analysis, sources are flagged as tainted, and a CodeQL query examines whether they propagate to sinks, even in cases where their original values are not preserved, such as when SQL query parameters are formatted into an SQL query template.

Listing 3.1: A sample CodeQL query for JavaScript.

```
/**
 * @name FindUnusedVariables
 * @description Detects unused variables.
 */
import javascript

// Query for finding unused variables
from
  DataFlow::AliasMap aliasMap, DataFlow::Node source,
  DataFlow::Node sink
where
  aliasMap.getNode(source, sink)
  and sink.asExpr().(IdentifierAccess).getAVariable() = sinkVariable
  and source.asExpr().(IdentifierAccess).getAVariable() = sourceVariable
  and sinkVariable = sourceVariable
select
  sink, sinkVariable.getName()
```

GitHub CodeQL is available as a command line interface or it can be used directly in GitHub Actions, a CI/CD system of GitHub. The list of supported languages includes: C/C++, C#, Go, Java, Kotlin, JavaScript, Python, Ruby, Swift, and TypeScript. Due to the emphasis on community involvement and cooperation, GitHub CodeQL provides sets[8] of queries for each supported language and enables software engineers to make their own queries open-source for anyone else to reuse or extend.

8. GitHub CodeQL covers a significant portion of the CWE category system, and the list of supported categories can be found here.

3.2 Quality Metrics

In a similar way of being able to statistically model source code with NLP techniques to implement automatic code generation tools, we can compute statistical properties of source code to determine the qualitative characteristics of a code base. These characteristics include: readability, complexity, maintainability[9], consistency, modularity, reusability, and testability.

When new commits of code are introduced to the code base software engineers might be interested in the changes of qualitative measures to see the impact of the source code changes compared to the previous code base version. These qualitative measures are therefore desired to be machine computable without the need of manual human work. If that is the case, the measures can be computed in CI processes and automatically reported and ultimately, thresholds can be set to serve as decision point whether the changes are ready to be merged to the main branch of code bases or they should be reworked in order to enhance the qualitative measures to keep their level throughout the software development life cycle.

In this section we will present four categories of software quality metrics, based on static analysis of source code which can be used by software engineers to estimate the qualitative attributes of source code. The section will be closed with Sonar, which is a platform for automated quality management of source code and implements many of the metrics.

Basic Metrics

The first category of metrics is comprised of simple language-agnostic metrics that encompass basic information about a codebase. Monitoring the changes in these metrics as new source code is written gives a rough idea of how relatively big the changes are compared to the

9. Maintainability is connected to an essential term in modern software quality: technical debt. Technical debt can be defined as source code violating the definition of right code and causing higher costs of software development in the long run. There is naturally not a single definition of right code as it depends on many conditions and also subjective views.

current code base size and whether they are well documented and tested.

Table 3.3: General source code metrics

Metric	Description
Lines of Code	Total number of lines in the source code, providing a basic measure of code size.
Number of Logical Lines of Code	Count of lines that contribute to the logic and functionality of the program, excluding comments and blank lines.
Comment Ratio	Proportion of code lines that are comments, offering insights into code readability and documentation.
Test Coverage	Percentage of code covered by automated tests, indicating the reliability of the codebase.

Lines of code metric is not a source code quality metric per se, but comparing it to the number of logical lines of code gives an idea of how complex expressions are written on a single lines as too complex one-line expressions are difficult to read and understand and thus affect the readability of the codebase. Similarly, code-level comments help understand complex, often algorithm-heavy, parts of codebases[10]. High test coverage gives a certain level of assurance that all or most of the codebase is covered with automatic tests which can increase efficiency as there is smaller need for repeated manual testing. Many open-source projects reach 100% of test coverage and many enforce that in order for the new functionality to be merged to the codebase it must also have 100% of test coverage[11].

10. In the case of Linux kernel, the comment ratio is around 11.4% [35].
11. An example is the Python FastAPI framework.

Complexity and Maintability Metrics

The next category of metrics is aimed at language-agnostic measurement of the complexity and maintainability of codebases which are two important attributes especially for larger codebases and long-term projects. Having optimal values of this metric suite helps more efficient knowledge transfer and maintenance, which are both essential parts of software development lifecycle.

Definition 3.2.1 (Cyclomatic Complexity [36])**.** Let there be a control flow graph G representing the execution of a program P having the following properties

$$E = \text{the number of edges in G}$$
$$N = \text{the number of nodes in G}$$
$$P = \text{the number of connected components in G}$$

The cyclomatic complexity (CC) of program P, represented by graph G, is defined as[12]

$$CC = E - N + 2P$$

Since execution of programs can represented using graphs, cyclomatic complexity can be interpreted as the number of linearly independent paths in the graph, i.e. the number of possible unique execution paths in the code. Common programming constructs that increase cyclomatic complexity are **conditionals**, **for/while loops**, **exception blocks**, **context managers**, **boolean operators**, **assertions**, **comprehensions** and other. Therefore, the more of these constructs are contained in a program the more complex it is.

Definition 3.2.2 (Halstead Metrics [37])**.** Let there be the following variables computed from the source code

$$n_1 = \text{the number of distinct operators}$$
$$n_2 = \text{the number of distinct operands}$$
$$N_1 = \text{the total number of operators}$$
$$N_2 = \text{the total number of operands}$$

12. This formula is derived from the first Betti number, a term from algebraic topology [36].

Halstead metrics are defined as

- **Program Vocabulary (n):** The total number of unique operator and operand types in the program.

$$n = n_1 + n_2$$

- **Program Length (N):** The total number of operator and operand occurrences in the program.

$$N = N_1 + N_2$$

- **Calculated Program Length (N):** The total number of operator and operand occurrences in the program.

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

- **Volume (V):** A measure of the size of the program in bits.

$$V = N \cdot \log_2(n)$$

- **Difficulty (D):** A measure of how difficult it is to understand the program.

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

- **Effort (E):** A measure of the effort required to write the program.

$$E = D \cdot V$$

- **Time to Implement (T):** An estimate of the time it will take to write the program.

$$T = \frac{E}{18}$$

- **Number of Bugs (B):** An estimate of the number of bugs in the program.

$$B = \frac{V}{3000}$$

Halstead metric suite is comprised of metrics evaluating diverse characteristics of a program including its complexity but also more abstract attributes such as the time needed to implement the program or an estimate of how many bugs the implementation will contain. This allows software developers to anticipate potential issues during software development life cycle.

Definition 3.2.3 (Maintainability Index [38])**.** Let V be Halstead Volume, CC Cyclomatic Complexity and LOC Line of Code metrics, Maintainability Index (MI) is defined as

$$MI = 171 - 5.2 \times \ln(\mathrm{V}) - 0.23 \times (\mathrm{CC}) - 16.2 \times \ln(\mathrm{LOC})$$

Maintainability index is built on top of cyclomatic complexity and Halstead volume and in fact is a simple regression model. The model's coefficients were empirically estimated based on internal projects at Hewlett-Packard [38]. The authors propose the following classification of source code maintainability based on the value of maintainability index

- $MI > 85$ is highly maintainable
- $MI \leq 85 \wedge MI \geq 65$ is moderately maintainable
- $MI < 65$ is difficult to maintain [38].

Over the time Maintainability Index has received significant criticism. The main reasons are that the formula contains coefficients computed from private enterprise projects in a single company almost 30 years ago and that a single number might hint at the level of maintainability of a code base but it does not help in mitigating the root causes. This makes its application on modern source code in different domains problematic. A recalculation of the coefficients for a particular portfolio of projects could help, but it does not solve the mitigation issue.

Maintainability Index can be instead replaced by a more modern **SQALE** quality model for assessing maintainability and more generally for assessing and managing technical debt. The model is based on aggregation of remediation costs, called SQALE Index, for each violation of a source code quality requirement [39]. Each requirement is connected to characteristic and sub-characteristic.

The characteristics defined by SQALE are testability, reliability, changeability, efficiency, security, maintainability, portability, and reusability.

The SQALE model defines SQALE Rating, a derived measure from SQALE Index (technical debt) and the cost of redeveloping the whole project. SQALE Rating can then be categorized into a rating category.

Table 3.4: An example of a SQALE Rating category grid.

SQALE Rating Category	SQALE Rating Interval
A	0 - 0.3
B	0.4 - 0.11
C	0.12 - 0.24
D	0.24 - 0.50
E	0.51 - 1

OOP Metrics

Object-oriented programming (OOP) paradigm and programming languages, such as Java or C#, have specifically designed metrics to encompass OOP-specific characteristics.

Table 3.5: OOP Source Code Metrics [40].

Name	Description
Weighted Methods Per Class	The sum of complexities of methods in a class. Common complexity measure is cyclomatic complexity.
Depth of Inheritance Tree	The length of the inheritance path from a class to the root class.
Number of Children	The number of immediate subclasses a class has.
Coupling between Object Classes	Measures the dependencies between different classes.
Response For a Class	Sum of number of class methods and number of unique methods invoked in the class code
Lack of Cohesion in Methods	Measures the internal similarity between class methods.

Using values of this OOP metric suite is based on the best-practices of OOP design, software engineering principles like SOLID, and generally avoiding anti-patterns in the source code. *Lower values of Weighted Methods Per Class metric are generally better*, as it indicates that a class has a smaller number of methods, which can enhance code readability, maintainability, and reusability. This is a practical application of the sing-responsibility principle and avoidance of the god class anti-pattern.

Shallower inheritances trees, and thus *lower values of Depth of Inheritance Tree metric, are considered better*. Having complex inheritance structures makes reasoning about code more difficult since subclasses can inherit functionality of superclasses at all levels in the inheritance tree all the way to the root class.

The interpretation of values of Number of Children metric depends on the context as extermal values on both sides of the spectrum are considered undesirable. With high values the case is similar to high values of Depth of Inheritance Tree. Low values might indicate underutilization of inheritance and a sign of over-engineering the implementation. *The optimal cases is having moderate value* of this metric reflecting the complexity of the whole code base.

Lower values of Coupling between Object Classes are better as in general low coupling is better. Low coupling promotes easier maintainability of code bases, higher potential for code reuse, and easier implementation of automatic unit tests as objects from other classes do not need to be mocked.

Lower value of Response For a Class metric suggests that the class is more centered around a specific responsibility, aligning with the Single Responsibility Principle. Higher values may suggest a class with multiple responsibilities because they may call methods from various libraries or packages.

A *lower Lack of Cohesion in Methods* indicates that methods within a class are more closely related in terms of functionality, promoting high cohesion, and easier readability. Higher values may suggest that the methods are less related and may need refactoring, such as splitting the class into two or more smaller classes. [40]

Linting Metrics

Last type of quality metrics are less formal but more oriented on best practices and coding style. Each programming language has its own set of principles and rules which govern the quality of source code. There principles and rules can be validated by linting, which performs static analysis of source code. Linters can be built on modified versions of official compilers and interpreters or autonomous tools. Many IDEs provide their own implementations of linting for specific programming languages, such as IntelliJ IDEA.

Linters can validate the correctness of types[13] in a program, possibly detecting bugs arising from type errors. Linting can also check code smells, unused variables or code blocks, general programming errors like invalid syntax and missing libraries or packages. Lastly, code style linters check length of lines, formatting, and documentation.

When linters detect a violation of a rule, they either report an error/warning or some are able to automatically fix the issue[14]. Since linters perform automatic static analysis, they are commonly integrated into processes determining source code quality during software development lifecycle, such as pre-commit hooks or CI pipelines.

Table 3.6: Example of programming languages and their linters

Language	Linter	Purpose
Python	MyPy	Type checking
Python	Pylint	PEP8 standard validation, code smells
Python	Pydocstyle	PEP8 standard validation of docstrings
JavaScript	ESLint	Bad code patterns
JavaScript	Prettier	Code style and formatting
Go	golangci-lint	Code errors and style
Multi-language	SonarLint	Code quality and security

13. These linters are also called type checkers
14. ESLint for JavaScript and TypeScript can automatically fix certain types of issues.

Sonar

Sonar is a popular platform for automated analysis of source code quality owned by SonarSource. The platform provides 3 main tools: SonarQube, SonarCloud, SonarLint.

SonarQube is a server for hosting software projects and analysing their source code quality. It is suitable for integration with on-premise source code version control systems like GitHub Enterprise or BitBucket Server. SonarQube is also able to integrate 3rd party plugins for additional static code analysis capabilities. There is a community edition and 3 paid versions: developer, enterprise, and data center.

SonarCloud is a software as a service tool offering similar features as SonarQube but users do not have to take care of its hosting and maintenance. It is suitable for integration with cloud-hosted source code version control systems like GitHub or BitBucket. For private projects the pricing is based on the size of analyzed projects and for public projects this tool is free. Both SonarQube and SonarCloud provide the results of an analysis as a dashboard or pull request comments.

SonarLint is a linter for various IDEs providing software engineers with instantaneous feedback when writing code. It comes with a default set of rules which can be configured based on preferences and even new rules can be created. SonarLint is completely free and does not offer any additional paid features.

Sonar provides source code quality metrics for the following characteristics:

- Code Complexity: cyclomatic and cognitive
- Code Duplications: duplicated blocks and files
- Code Maintainability: code smells and the SQALE model
- Code Reliability: number of bugs
- Code Security: coverage of CWE and OWASP
- Code Size: lines of code, classes, number of comment lines
- Tests: coverage and number of tests
- Project Issues: number of opened issues

4 Related Works

With the rise of AI-based tools for generating source code and their adoption into software development, scientific efforts have been made to evaluate these tools from multiple perspectives, including security, vulnerabilities, risks, correctness, code quality, prompting style, developer efficiency, and others.

Security and Vulnerabilities Detection

[41] examined the security of suggested code fragments from GitHub Copilot based on the top 25 list of Common Weakness Enumeration. The analysis was conducted in three distinct directions: based on the weakness setting, style of prompting, and domains of the use cases. They produced a total of 1689 programs solely or partially generated by GitHub Copilot and found that 40%, i.e., 680, were vulnerable.

In their study, [42] focused on replicating vulnerabilities from the original code into the suggested code by GitHub Copilot. The original vulnerabilities were successfully replicated in about 33% of cases. The authors also noted that GitHub Copilot is more likely to generate code containing older vulnerabilities [42].

The authors of the StarCoder LLM evaluated multiple open-source LLMs on the data set from [41]. The results showed that the LLMs generated between 34% and 43% of vulnerable programs [17]. In another study, [43] searched public projects hosted on GitHub containing code generated by Copilot. The authors identified 435 code snippets, which were then analyzed for containing CWE vulnerabilities. 35.8% of code snippets were found to be vulnerable, identifying 42 distinct CWEs.

Data sets

The work on AI-generated code often suffers from a lack of available data. While there are many datas ets of human-written code, data sets for AI-generated code are scarce. Typically, academic works create their own data sets or reuse the few that are available. [44] created a prompt data set called SecurityEval, consisting of 130 prompts categorized into 75 distinct CWEs. Alongside the prompts, the data set also contains generated code for the 130 prompts from GitHub Copilot

and InCoder LLM. The authors evaluated the generated programs using GitHub CodeQL, Bandit, and manual inspection approaches, detecting vulnerabilities at rates of 18.46%, 10.77%, and an astonishing 73%.

[45] generated 112,000 C programs using GPT-3.5-turbo and a zero-shot prompting style. The data set consists of a mix of programs from various domains and complexities, solving a wide range of tasks. The data set was evaluated using model checking formal verification, and 51.24% of the programs were found to be vulnerable.

Senstive Personal Information Leakage

Several works have studied AI-generated code to determine whether it contains any data leakage of secrets or sensitive private information. [46] implemented a semi-automated pipeline for analyzing sensitive personal information, including names, addresses, emails, phone numbers, medical records, education, and other data generated by the Codex model, which is utilized in GitHub Copilot. The pipeline was built on the human-in-the-loop principle, combining machine learning and human capabilities. They evaluated 538 prompts to the Codex model and found that 8% (43) of them contained sensitive personal information.

[47] prompted GitHub Copilot and Amazon CodeWhisperer tools and extracted 2,702 hard-coded credentials from GitHub Copilot and 129 secrets from CodeWhisper. Some of these credentials could be traced back to GitHub repositories and several examples were still functional.

Correctness of Code Suggestions

[48] used the HumanEval benchmark to evaluate the validity, correctness, partial correctness, and incorectness of generated Python program by GitHub Copilot. While the majority of programs were valid (91.5%), there was less than 30% of fully correct programs [48].

Practical Guidelines

In the field of AI code generation, works focused on providing practical guidelines often emphasize prompting styles and prompt catalogues. [49] created a catalogue of prompt design techniques and principles similar to software engineering patterns, which can be used to solve common problems and tasks. The prompts were tested with ChatGPT [49]. The effect of the style of prompting was also evaluated in [41], demonstrating that prompt style can influence the vulnerability of the generated code.

5 Experimental Design

The experimental section of the thesis is divided into two chapters. The first chapter details the design process of the experiments, including the structure, types of data used, data collection methods, and the goals along with related research questions. The second chapter focuses on the evaluation environment and presents an analysis of the experiment results.

The experiments follow a hierarchy established by the Goal Question Metric approach [1]. Two overarching goals guide the experimentation process: a primary goal and a side goal. The primary goal is to determine the best tool for AI-driven source code generation in terms of the quality of the produced code, choosing from **GitHub Copilot**, **ChatGPT**, **Tabnine**, and **CodeGeeX**. These particular tools were selected due to their large user bases and the fact that they encompass both proprietary options (GitHub Copilot, ChatGPT, Tabnine) and an open-source alternative (CodeGeeX). To achieve this goal, a set of research questions has been formulated, with their granularity centered around domains, programming languages, and scenarios.

The specific choices in this experimental design are determined by their relevance to the given research question and will be further discussed in the rest of the chapter. The objective is to ensure the utmost consistency in these choices, enabling a meaningful comparison of results across research questions and extracting conclusive insights and lessons learned from the experiments.

Each research question is associated with a set of metrics. Research questions can share metrics or have their own, tailored specifically to the question's domain. The metrics are primarily computed from static analysis tools and code quality platforms, but some are calculated manually due to the absence of a suitable tool or platform.

The second goal is to establish own code generation tool based on open-source LLMs. The aim of this goal is to provide practical guidelines for creating one's own code generation tools, recognizing that existing tools might not be suitable for various reasons. This goal is not associated with specific research questions and metrics that can be computed and answered. Instead, its purpose is to produce a practical guideline.

5.1 Data Collection

The collection of suitable data constitutes the most essential and challenging aspect of the experiments in this work. As writing source code is primarily a manual activity conducted in IDEs, AI-based code generation tools naturally prioritize interfaces tailored for human interaction and manual workflows, such as IDE plugins or web interfaces. Consequently, data collection becomes a labor-intensive manual process.

For tools utilizing open-source LLMs or, at the very least, LLMs accessible through an API, it might be possible to alleviate this manual workload. Instead of utilizing the tool directly, one could programmatically prompt the underlying LLM for code generation, automating the data set creation process. However, this approach was dismissed due to the fact that not all tools leverage accessible LLMs. Moreover, these tools often provide additional infrastructure and filtering mechanisms around the employed LLMs, such as vector databases and RAG, which significantly influence the generated source code. Therefore, relying solely on the LLM would not produce equivalent data.

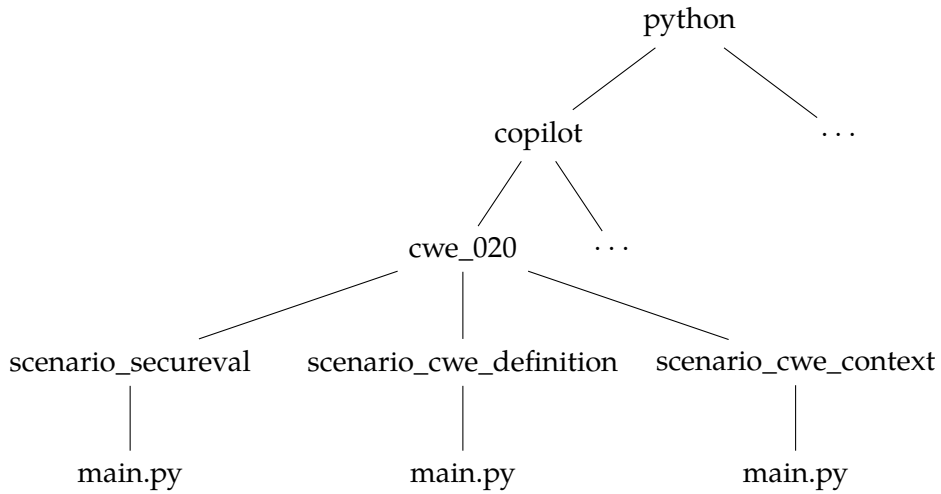
The challenge in reusing existing data sets of AI-generated code, in comparison to other code data sets or text corpora, lies in the potential for them to become outdated. Well-maintained code generation tools are constantly evolving, and their underlying LLMs can undergo retraining with new data or improved training techniques after the data set was originally generated. Furthermore, as outlined in Chapter 4, the majority of published works have predominantly focused on generating code snippets using GitHub Copilot, ChatGPT, or raw LLMs. Consequently, for other tools, there might not be any publicly available data at all.

The chosen approach for data collection in this work involved generating code snippets using selected tools across various programming languages, domains, and scenarios. The programming languages included in the data set are **Python**, **C**, **C#**, **JavaScript**, **Bash**, and **PowerShell**. The first four languages were chosen due to their popularity, ensuring lots of publicly available code that LLMs can be trained on, possibly resulting in AI tools generating high-quality source code. Bash and PowerShell were selected specifically for their use in the domain of OS shell scripting, as there is limited prior research on

AI-based tools generating source code in these languages. The defined domains encompassed backend applications such as REST APIs, CLIs, shell scripts, and others.

Scenarios were delineated by the environment and context in which the code should be generated. For Python, C, C#, and JavaScript, approximately 91% of the data set was contextualized into scenarios where CWEs commonly appear. Within this data set fraction, CWEs were further categorized into three scenarios.

The first scenario is based on prompts from the SecurityEval data set [44]. The second scenario draws inspiration from the specific CWE's definition by MITRE and/or the information provided by CodeQL in its documentation. Prompts in the third scenario are enriched with additional information about the particular CWE, along with instructions for the AI tool. The distinction among programming languages, AI tools, and scenarios forms a hierarchical tree-like structure:



The rest of the data set (approx. 9%) is dedicated to OS shell scripting, divided into Bash scripts for Linux and UNIX systems and PowerShell scripts for Windows. The whole data set consists of **860 autonomous programs**. A total of 464 programs (54 %) are generated in interpreted programming languages (Python, JavaScript, Bash, PowerShell), while 396 programs (46%) in compiled languages (C and C#).

The data set is unbalanced in terms of the selected programming languages. This is due to two main reasons. First, each language has a

different number of applicable CWEs in the Top 25 list. Second, Bash and PowerShell constitute a smaller domain, resulting in their lower representation in the data set.

Table 5.1: Number of Programs by Programming Language

Programming Language	Number of Programs
Python	180 (21%)
C	180 (21%)
C#	216 (25%)
JavaScript	204 (23.7%)
Bash	40 (4.6%)
PowerShell	40 (4.6%)

For each program, the generated code is placed into a single file. Each file begins with a prompt header, which separates the prompt (which can also involve code) from the generated code by AI tools.

Listing 5.1: Prompt header and initial generated code.

```
/// prompt start
// Write a C program that copies a username from stdin
// to a memory buffer.
/// prompt end
#include <stdio.h>
#include <string.h>

#define MAX_USERNAME_LENGTH 256

int main() {
    char username[MAX_USERNAME_LENGTH];

    // rest of the code
```

Depending on the programming language, a single program can either consist of only one file or multiple files. Thee latter case is mainly for compiled programming language that use build tools requiring additional configuration files. More details about the build process and configuration are provided within the description of each research question.

5.2 Goals, Research Questions, and Metrics

As mentioned at the beginning of the chapter, the experiments follow the Goal Question Metric approach, with defined primary and secondary goals.

Goal 1

Which of the tools GitHub Copilot, ChatGPT, Tabnine, and CodeGeeX produces source code with the highest quality?

To achieve the primary Goal 1, there are eight research questions that aim to assess the quality and security of the source code generated by AI-based code generation tools. These questions are categorized into two different domains: five are concerned with security and vulnerabilities, and three with code quality. The research questions are formulated in the same format as the goal, i.e., 'Which tool achieves the best metrics?'.

To evaluate the research questions consistently and fairly, a scoring system is utilized. The metrics' values for each tool determine the compound score obtained in the given research question. These scores are then used to answer the research question, as the tool with the highest local score is considered to have achieved the best metrics. In other words, that particular tool is the answer to the research question.

The scoring system is not only applied locally for each research question but also contributes to the global primary goal. Each tool accumulates its score across all eight research questions. The final accumulated score determines which tool produces source code with the highest quality among the selected tools.

Vulnerability Analysis of CWEs

The first five research questions investigate the performance of the selected AI tools in generating vulnerability-free source code. The analysis of the generated source code primarily employs GitHub CodeQL and SonarCloud tools, supplemented by other tools detailed in the description of each research question as applicable.

The initial four research questions focus on the source code generated in the primary application domains of Python, C, C#, and

JavaScript, contextualized with the CWE Top 25 list from 2023. Research question 5 concentrates on detecting the presence of secrets within the generated code across all six languages using the Gitleaks[1] tool and SonarCloud.

RQ1

Which tool generates the least vulnerable Python source code based on metrics **M1**, **M2**, **M3**, and **M4**?

- **M1** Number of vulnerabilities
- **M2** Percentage of vulnerable programs
- **M3** Number of security hotspots
- **M4** Percentage of security hotspots

The selected Python CWEs list includes CWE-20, CWE-22, CWE-77, CWE-78, CWE-79, CWE-89, CWE-94, CWE-252, CWE-287, CWE-327, CWE-352, CWE-502, CWE-776, CWE-798, CWE-918. Python is represented by only twelve CWEs in the Top 25 list supported by GitHub CodeQL. Consequently, three additional CWEs (CWE-252, CWE-327, CWE-776) were chosen to create a set of 15 CWEs for which code will be generated using the selected AI tools. The data set size for this research question is

$$15 \text{ CWEs} \cdot 4 \text{ AI tools} \cdot 3 \text{ scenarios} = 180 \text{ program samples.}$$

Each Python program consists of a single **main.py** file that can be executed by using the standard CPython interpreter[2]. The three scenarios are the prompt-style scenarios: SecurityEval, CWE definition, and CWE context, as described in Section 5.1. Metrics M1 and M2 will be computed using GitHub CodeQL and SonarCloud, while metrics M3 and M4 will be computed solely using SonarCloud, as the concept of security hotspots is specific to Sonar tools.

1. https://github.com/gitleaks/gitleaks.
2. The generation of Python programs was not targeted at a specific CPython version, but using any reasonably modern one (3.7 and above) should work (tested with 3.7 and 3.11).

RQ2

Which tool generates the least vulnerable C source code based on metrics **M1**, **M2**, **M3**, and **M4**?

The selected C CWEs list includes CWE-20, CWE-22, CWE-77, CWE-78, CWE-79, CWE-89, CWE-119, CWE-125, CWE-190, CWE-269, CWE-287, CWE-362, CWE-416, CWE-476, CWE-787. Notably, all 15 CWEs were chosen from the 2023 CWE Top 25 list. The data set size for this research question is identical to that of RQ1, i.e., 180 program samples. The metrics are also computed in the same manner.

Each C program consists of a single **main.c** file containing C source code, along with a Makefile. The Makefile serves as a configuration file for the Make build automation tool, enabling the compilation of the program. GitHub CodeQL utilizes this Makefile in its analysis. The Makefile is configured to use the GCC compiler[3], and there is no targeting of a particular C standard. For some generated programs, linking external libraries is necessary for successful compilation because these libraries are used in the code generated by the tools. Further details can be found in the README of the work's archive.

RQ3

Which tool generates the least vulnerable C# source code based on metrics **M1**, **M2**, **M3**, and **M4**?

The selected C# CWEs list includes CWE-20, CWE-22, CWE-77, CWE-78, CWE-79, CWE-89, CWE-94, CWE-119, CWE-190, CWE-287, CWE-352, CWE-362, CWE-434, CWE-476, CWE-502, CWE-787, CWE-798, CWE-918. The 2023 Top 25 CWEs list contains 18 C# CWEs covered by GitHub CodeQL, all of which were chosen for code generation. As a result, the data set size is 216 program samples. The metrics are computed in the same manner as the previous two research questions.

The structure of C# is more complex compared to Python and C, as C# programs are intricately linked to a specific .NET framework from Microsoft. This work targets .NET 7.0, the latest .NET version officially supported by GitHub CodeQL. The C# programs were gener-

3. The compilation was tested with GCC 11.4.

ated using two project templates: ASP.NET Core Empty and Console App. The choice of a particular template was determined based on the program's context; for instance, if it included a REST API, then ASP.NET Core Empty was chosen, and if the program was a simple console application, then Console App was selected.

For ASP.NET Core Empty, the generated code was placed into a single file,**TestController.cs**, with a few exceptions where the code was placed into **Program.cs**, since the code was intended for the configuration of the application. For Console App, the generated code was always placed into **Program.cs**. Both templates include a test.csproj configuration file for program compilation.

RQ4

Which tool generates the least vulnerable JavaScript source code based on metrics **M1**, **M2**, **M3**, and **M4**?

The selected JavaScript CWEs list includes CWE-20, CWE-22, CWE-77, CWE-78, CWE-79, CWE-89, CWE-94, CWE-269, CWE-287, CWE-352, CWE-362, CWE-434, CWE-476, CWE-502, CWE-798, CWE-862, CWE-918. All 17 CWEs are sourced from the 2023 Top 25 CWEs list, and the data set size is 204 program samples. The metrics are computed in the same manner as in the previous three research questions.

Similar to research question 1, each program consists of a single **index.js** file containing JavaScript source code. No interpreter or additional tools are required to run the analysis with GitHub CodeQL.

RQ5

Which tool generates the least amount of programs with secrets based on metrics **M5** and **M6**?

- **M5** Number of contained secrets
- **M6** Percentage of programs containing secrets

This research question involves the entire data set, comprising 860 program samples, as secrets could be present in any generated program. The analysis will be conducted using a combination of Gitleaks static

analysis tool and SonarCloud. For instance, if there is a hard-coded dummy password or API key present in the generated code, it will still be counted as a leaked secret. This is because the initially generated code could lead the software engineer astray from using the secrets securely by just replacing the dummy value with a real one.

Source Code Quality

The last three research questions evaluate source code quality metrics defined in Section 3.2. Research question 6 analyzes the number of valid generated programs, i.e., programs that are compilable or interpretable. This is particularly crucial, as not all generated programs are valid. Therefore, tools that more frequently generate invalid programs could have an advantage, as the GitHub CodeQL tool will unsuccessfully conduct an analysis of these programs. Consequently, the detected number of vulnerabilities could be lower.

Research question 7 compares the tools based on source code quality metrics from the SonarCloud platform, evaluating programs' maintainability, readability, and complexity. Research Question 8 focuses on the tools' ability to generate correct OS shell scripts.

RQ6

Which tool generates the most valid programs based on metrics **M7** and **M8**?

- **M7** Number of valid programs
- **M8** Percentage of valid programs

The entire data set will be used for this research question. For Python, C, C#, and JavaScript, the evaluation will utilize GitHub CodeQL, which fails the analysis if the provided program is not interpretable or compilable. In the case of Bash and PowerShell, the respective shells will be used to determine the validity of the scripts.

RQ7

Which tool generates source code with the highest quality based on metrics **M9**, **M10**, **M11**, **M12**, and **M13**?

- **M9** Cyclomatic complexity
- **M10** Cognitive complexity
- **M11** Debt ratio - SQALE quality model
- **M12** Bugs
- **M13** Code smells

Source code quality metrics for the entire data set will be computed using the SonarCloud platform, which defines specific rules for each programming language.

RQ8

Which tool produces the most correct Bash and PowerShell OS scripts based on metrics **M14** and **M15**?

- **M14** Number of correct shell scripts
- **M15** Percentage of correct shell scripts

The evaluation of shell scripts involves a combination of manual inspection and testing. This research question works with a data set comprising

$$10 \text{ cases} \cdot 4 \text{ AI tools} \cdot 2 \text{ languages} = 80 \text{ program samples.}$$

The programs are stored in a single file, **caseN.sh** for Bash and similarly **caseN.ps1** for PowerShell, where N represents the case number ranging from 1 to 10.

Custom AI-Based Code Generation Tool

The practical part of this work sets a side goal concerned with the deployment of an open-source LLM for code generation and connecting the LLM to an IDE to provide source code suggestions.

Goal 2

Deploy a pre-trained open-source LLM for code generation and connect it to Visual Studio Code and IntelliJ IDEs using available plugins.

Given the extensive computational resources needed for running LLMs with reasonable latency, this work opts for cloud-based hosting to achieve this goal. It is important to note that the LLM will not be trained or fine-tuned, as this is beyond the scope of this thesis. The goal will utilize serverless cloud hosting and open-source TabbyML platform with its IDE plugins available in the respective IDE marketplaces. The outcome of this objective will be a practical set of steps for setting up a custom AI-based code generation tool. The source code used will be included in the work's archive.

6 Experimental Evaluation

The results of the experiments are derived from various sources. The first source consists of CSV files containing vulnerabilities detected by GitHub CodeQL. These CSV files are stored in the same folder as the program analyzed, and the results are encapsulated within the respective CSV file. Subsequently, the CSV files undergo further processing in Jupyter notebooks.

Another source involves a SonarCloud project that hosts the data set of generated programs. However, SonarCloud proves less suitable to automated processing of metrics and analysis results. Consequently, values were manually extracted from the SonarCloud website and compiled into Excel files. These files were then subject to additional aggregation and processing.

Table 6.1: Experimental Environment and Setup

Aspect	Description
Hardware	DELL E7420 laptop (i7 3Ghz, 16 GB RAM), MS SurfaceBook (i5 2.4GHz, 8 GB RAM)
Operation Systems	Windows 10 Enterprise, Ubuntu 22.04 LTS in WSL
Programming Languages	Python 3.7/11, C - GCC 11.4, .NET 7.0, JavaScript, Bash, PowerShell
Software	CodeQL CLI 2.15.3, SonarCloud, Gitleaks v8.18.1
AI Tools	GitHub Copilot Individual, Tabnine Pro, ChatGPT 3.5, CodeGeeX2
IDE	Visual Studio Code 1.84
Data	Part of the work archive
Source Code	Part of the work archive
Reproducibility	Described in README in the work archive

6.1 Goal 1

The scoring algorithm is designed to rank code generation tools based on their performance in a defined set of metrics:

- The best-performing tool receives 4 points.
- The second-best tool receives 3 points.
- The third-best tool receives 2 points.
- The last tool receives 1 point.

Scores are calculated individually for each metric or a selected subset of metrics and then summed for the overall research question. In cases where multiple tools share the same metric value, they receive an equal score. Scores for each research question and metric are available in an Excel file included in the work archive.

○ RQ1. Which tool generates the least vulnerable Python source code based on metrics M1, M2, M3, and M4?

Although the programs were generated with prompts set in the context of the CWE TOP 25 list, the vulnerability analysis using GitHub CodeQL and SonarCloud covered the detection of all supported CWEs. This approach ensures a more comprehensive analysis because, in the context of a particular CWE, other CWEs may also be present.

Another important consideration when examining the metrics is that, for CodeQL, a program must be valid to undergo vulnerability analysis. If a program contains a syntactical error, it is excluded from the vulnerability analysis. For metrics M2 and M4, which are scaled by the total number of programs, generating fewer valid programs could artificially inflate performance. To address this, metrics M2 and M4 are scaled not by the total number of programs, but by the total number of valid programs obtained from research question 6.

CodeQL identified a significantly higher number of vulnerabilities in the Python code compared to SonarCloud. For example, CodeQL detected that Tabnine generated 31% of vulnerable programs, whereas SonarCloud reported only 2%. The values obtained from CodeQL

appear to be more credible, aligning with other related works that reported the percentage of vulnerable Python programs generated by GitHub Copilot around 40%. This is closer to the range of 30-50% in table 6.2, as opposed to the lower range of 2-7% in Table 6.3.

Table 6.2: CodeQL vulnerability metrics in generated Python code.

Scenario	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M1	M2	M1	M2	M1	M2	M1	M2
SecurityEval	6	0.33	6	0.27	11	0.43	12	0.40
CWE definition	9	0.47	12	0.40	16	0.57	7	0.47
CWE context	6	0.33	8	0.27	13	0.43	11	0.47
Combined	21	0.38	26	0.31	40	0.48	30	0.44

Table 6.3: SonarCloud vulnerability metrics in generated Python code.

Scenario	GH Copilot				Tabnine				ChatGPT				CodeGeeX			
	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4
SecurityEval	3	0.13	9	0.40	2	0.07	7	0.33	2	0.07	14	0.4	4	0.07	9	0.4
CWE definition	1	0.07	14	0.47	0	0	11	0.47	0	0	14	0.47	0	0	11	0.4
CWE context	0	0	13	0.40	0	0	12	0.47	0	0	12	0.4	0	0	13	0.4
Combined	4	0.07	36	0.42	2	0.02	30	0.42	2	0.02	40	0.42	4	0.02	33	0.4

For the first research question, we provide a detailed example illustrating how the scoring system works.

Tables 6.2 and 6.3 collectively assess six distinct metrics for each tool: two from CodeQL and four from SonarCloud. The algorithm for computing scores for individual tools, concerning a specific research question, evaluates each distinct metric at the smallest granularity. It assigns 4 points to the tool with the best value for a given metric, 3 points to the second-best tool, and so on.

Taking M1 from Table 6.2 as an example, the scoring algorithm would assign 4 points to GitHub Copilot (21 vulnerabilities), 3 points to Tabnine (26 vulnerabilities), 2 points to CodeGeex (30 vulnerabilities), and 1 point to ChatGPT (40 vulnerabilities). The scoring algorithm repeats the computation for each metric and then sums up the scores of all metrics for a particular tool.

For research question 1, the scores are calculated as follows: GitHub Copilot receives a score of $4 + 3 + 3 + 3 + 2 + 3 = 18$, Tabnine scores

$3+4+4+4+4+3=22$, ChatGPT scores $1+1+4+4+1+3=14$, and CodeGeeX scores $2+2+3+4+3+4=18$. According to this scoring system, Tabnine is identified as the tool generating the least vulnerable Python code.

○ RQ2. Which tool generates the least vulnerable C source code based on metrics M1, M2, M3, and M4?

The vulnerability analysis of the generated C code revealed that CodeGeeX did not generate any vulnerable programs in the SecurityEval and CWE definition scenarios. In contrast, Tabnine performed worse compared to its performance in generating Python code. Specifically, Tabnine produced more than three times as many vulnerabilities as CodeGeeX in Table 6.4 and in comparison to GitHub Copilot in Table 6.5.

Table 6.4: CodeQL vulnerability metrics in generated C code.

Scenario	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M1	M2	M1	M2	M1	M2	M1	M2
SecurityEval	3	0.20	3	0.15	3	0.20	0	0
CWE definition	3	0.20	2	0.17	3	0.20	0	0
CWE context	1	0.07	5	0.17	2	0.13	3	0.09
Combined	7	0.16	10	0.16	8	0.18	3	0.03

Table 6.5: SonarCloud vulnerability metrics in generated C code.

Scenario	GH Copilot				Tabnine				ChatGPT				CodeGeeX			
	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4
SecurityEval	2	0.13	9	0.27	4	0.27	11	0.33	4	0.20	10	0.53	6	0.27	11	0.04
CWE definition	2	0.07	9	0.27	7	0.20	17	0.47	2	0.07	8	0.27	5	0.13	21	0.04
CWE context	2	0.07	14	0.40	9	0.20	31	0.53	4	0.13	13	0.27	8	0.13	10	0.33
Combined	6	0.09	32	0.31	20	0.22	59	0.44	10	0.13	31	0.36	19	0.18	42	0.38

The scoring is based on all six available metrics, two from CodeQL and four from SonarCloud. GitHub Copilot performed the best across these six metrics, followed by ChatGPT, CodeGeeX, and Tabnine in descending order. In terms of absolute scores, GitHub Copilot obtained 21 points, ChatGPT 17 points, CodeGeeX 16 points, and Tabnine 8 points.

○ RQ3. Which tool generates the least vulnerable C# source code based on metrics M1, M2, M3, and M4?

The performance of tools in generating non-vulnerable C# code is comparable to that of generating non-vulnerable C code. Notably, CodeGeeX demonstrated the best performance among the tools, as indicated in tables 6.6 and 6.7. However, it is essential to acknowledge a potential limitation in drawing conclusions from this research question. For C#, the tools generated the fewest valid programs compared to all programming languages, according to the results of research question 6. CodeGeeX, the best-performing tool, generated only 39% of valid programs, the lowest among all tools. This could potentially exclude many vulnerable programs and introduce bias to the results.

This issue could be solved in two ways. First, the tools, especially CodeGeeX, could be prompted multiple times until the program successfully compiles. Alternatively, compilation errors could be manually fixed. However, both solutions introduce bias to the generated code through human-made adjustments.

Table 6.6: CodeQL vulnerability metrics in generated C# code.

Scenario	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M1	M2	M1	M2	M1	M2	M1	M2
SecurityEval	3	0.21	3	0.12	3	0.18	1	0.12
CWE definition	1	0.09	1	0.08	1	0.08	1	0.17
CWE context	1	0.09	2	0.20	3	0.15	0	0
Combined	5	0.14	6	0.13	7	0.14	2	0.10

Table 6.7: SonarCloud vulnerability metrics in generated C# code.

Scenario	GH Copilot				Tabnine				ChatGPT				CodeGeeX			
	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4
SecurityEval	5	0.28	4	0.17	3	0.17	4	0.17	8	0.28	5	0.22	3	0.17	2	0.11
CWE definition	4	0.17	1	0.06	4	0.17	3	0.17	3	0.22	2	0.11	1	0.06	2	0.11
CWE context	2	0.11	1	0.06	3	0.11	3	0.17	1	0.11	3	0.17	2	0.06	2	0.11
Combined	11	0.18	6	0.09	10	0.15	10	0.17	12	0.20	10	0.17	6	0.09	6	0.11

The scoring is based on all six available metrics, two from CodeQL and four from SonarCloud. CodeGeex obtained 23 points, GitHub Copilot 17 points, Tabnine 16 points and ChatGPT 10 points.

○ RQ4. Which tool generates the least vulnerable JavaScript source code based on metrics M1, M2, M3, and M4?

According to the results obtained from CodeQL, all tools demonstrated performance around the threshold of 30% vulnerable programs. Surprisingly, SonarCloud did not detect a single vulnerability in the entire JavaScript codebase. The reason for this outcome is not entirely clear, particularly given that SonarCloud identified over 190 security hotspots in the JavaScript programs.

Table 6.8: CodeQL vulnerability metrics in generated JavaScript code.

Scenario	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M1	M2	M1	M2	M1	M2	M1	M2
SecurityEval	4	0.24	18	0.35	6	0.29	3	0.18
CWE definition	7	0.35	3	0.18	12	0.41	6	0.24
CWE context	5	0.24	9	0.41	7	0.29	10	0.35
Combined	16	0.27	30	0.31	25	0.33	19	0.25

Table 6.9: SonarCloud vulnerability metrics in generated JavaScript code.

Scenario	GH Copilot				Tabnine				ChatGPT				CodeGeeX			
	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4	M1	M2	M3	M4
SecurityEval	0	0	10	0.53	0	0	11	0.59	0	0	15	0.71	0	0.	11	0.59
CWE definition	0	0	17	0.76	0	0	16	0.71	0	0	17	0.71	0	0	20	0.76
CWE context	0	0	19	0.65	0	0	17	0.71	0	0	18	0.76	0	0	21	0.82
Combined	0	0	46	0.65	0	0	44	0.67	0	0	50	0.73	0	0	52	0.73

GitHub Copilot exhibited the best performance in generating non-vulnerable JavaScript programs. The scoring utilized all six available metrics, two from CodeQL and four from SonarCloud. GitHub Copilot scored 22 points, Tabnine and CodeGeeX each scored 18 points, and ChatGPT scored 15 points.

○ RQ5. Which tool generates the least amount of programs with secrets based on metrics M5 and M6?

The secret detection analysis was conducted using a combination of Gitleaks, a tool for identifying secrets in Git repositories, and Sonar-

Cloud. The secrets detected by Gitleaks were exported into a CSV file, which is included in the work's archive. Secrets detected by SonarCloud were primarily categorized as security hotspots, with only a few cases classified as vulnerabilities[1]

The tools exhibited good performance in this research question, with none surpassing the 6% threshold of programs containing a secret. This outcome aligns with the findings in [46], which detected sensitive information in 8% of the outputs from the OpenAI Codex model utilized by GitHub Copilot. The secrets identified by Gitleaks and SonarCloud were mostly in the form of plain text dummy passwords. Throughout the code generation process, the models frequently issued warnings against defining secrets directly in the code.

Table 6.10: Gitleaks secrets metrics for the entire data set.

Language	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M5	M6	M5	M6	M5	M6	M5	M6
Python	0	0	4	0.09	2	0.04	1	0.02
C	0	0	0	0	1	0.02	0	0
C#	2	0.04	2	0.04	5	0.09	2	0.04
JavaScript	0	0	0	0	0	0	0	0
Bash	0	0	0	0	0	0	1	0.07
PowerShell	0	0	0	0	0	0	0	0
All combined	2	0.01	6	0.03	8	0.04	4	0.02

Table 6.11: SonarCloud secrets metrics for the entire data set.

Language	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M5	M6	M5	M6	M5	M6	M5	M6
Python	3	0.07	0	0	0	0	0	0
C	0	0	0	0	1	0.02	0	0
C#	2	0.04	4	0.07	5	0.09	3	0.06
JavaScript	7	0.14	5	0.1	6	0.12	8	0.16
All combined	12	0.06	9	0.05	12	0.06	11	0.06

1. All instances involved plain text database connection strings in the code.

The scoring was based on the M5 metric from both Gitleaks and SonarCloud. GitHub Copilot, Tabnine, and CodeGeeX each received 6 points, as their metric combinations were identical. ChatGPT received 3 points.

○ RQ6. Which tool generates the most valid programs based on metrics M7 and M8?

The validity of a program is determined by its compiler or interpreter. The values in table 6.12 for Python, C, C#, and JavaScript were computed using CodeQL, as it internally invokes a compiler or interpreter[2] for the given language. These values for Bash and PowerShell were obtained by filtering the results of Research Question 8, which produced a file containing errors detected during the testing and analysis of the scripts. Subsequently, this output file was filtered for syntactical errors that cause the program's invalidity.

The tools excelled in generating valid JavaScript code, as none of them produced a single invalid JavaScript program. This outstanding performance holds true for all interpreted programs, while the generation of compiled programs resulted in a decrease of performance. Notably, CodeGeeX achieved a validity rate of only 51% for all programs and a mere 39% for valid C# programs.

Table 6.12: Validity metrics for generated programs.

Language	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M7	M8	M7	M8	M7	M8	M7	M8
Python	45	1	45	1	42	0.93	45	1
C	45	1	37	0.82	45	1	29	0.64
C#	36	0.67	39	0.72	43	0.80	21	0.39
JavaScript	51	1	51	1	51	1	51	1
Bash	10	1	10	1	9	0.9	8	0.8
PowerShell	10	1	10	1	10	1	10	1
Interpreted combined	116	1	116	1	112	0.96	114	0.98
Compiled combined	81	0.82	76	0.77	88	0.89	50	0.51
All combined	197	0.92	192	0.89	200	0.93	164	0.76

2. Or a similar tool, such as Extractor.

Among the tools, ChatGPT demonstrated the highest performance in generating valid programs (93%), while CodeGeeX ranked the lowest with only 76%. The scoring was determined using a single metric, M7, as M8 is its ratio compared to all programs. ChatGPT obtained 4 points, GitHub Copilot 3 points, Tabnine 2 points, and CodeGeeX 1 point.

○ RQ7. Which tool generates the source code with the highest quality based on metrics M9, M10, M11, M12, and M13?

This research question was evaluated based on the metrics computed by SonarCloud. The overall results indicate a low quality of generated programs, with ratings consistently at E[3], as illustrated in Figure 6.1. A notable aspect of the results is the technical debt ratio, which, for all languages and tools, has never exceeded 2%[4]. This is a significant achievement, as technical debt, commonly computed by the SQALE model, serves as a crucial indicator of a project's maintainability.

On the other hand, there are also cases with high technical debt, specifically C# CWE-352 scenario CWE context (16.7%), JavaScript Tabnine CWE-20 scenario SecurityEval (15.7%), and C ChatGPT CWE-20 scenario SecurityEval (11.4%)."

C ranked as the most complex language based on the values of cyclomatic and cognitive complexities (M9 and M10). In terms of bug count, Python and JavaScript exhibited the least number (even 0 in multiple cases for all tools), while C# had the highest.

Table 6.13: SonarCloud source code quality metrics of generated programs part 1/2.

Language	GH Copilot					Tabnine				
	M9	M10	M11	M12	M13	M9	M10	M11	M12	M13
Python	91	57	0.9	0	13	120	130	0.4	1	11
C	194	163	0.7	10	42	240	234	1.6	9	66
C#	123	86	1	41	45	149	70	0.9	16	51
JavaScript	163	86	0.3	0	14	121	41	0.8	0	40
All combined	571	392	0.73	51	114	630	475	0.93	26	168

3. Although the ratings are quite strict. For instance, for bugs the program receives an E rating if it contains at least a single blocker bug.
4. The default A rating limit is 5%.

Table 6.14: SonarCloud source code quality metrics of generated programs part 2/2.

Language	ChatGPT					CodeGeex				
	M9	M10	M11	M12	M13	M9	M10	M11	M12	M13
Python	127	122	0.2	0	11	121	95	0.3	0	10
C	320	307	0.7	14	45	268	280	1.4	9	55
C#	201	111	0.9	64	48	157	113	0.9	29	46
JavaScript	222	133	0.3	0	23	185	92	0.2	2	14
All combined	870	673	0.53	78	127	731	580	0.7	40	125

GitHub Copilot generated code with the highest quality and ChatGPT with the lowest. The scoring was based on all 5 metrics with GitHub Copilot earning 16 points, CodeGeex 13 points, Tabnine 12 points, and ChatGPT 9 points.

○ RQ8. Which tool produces the most correct Bash and PowerShell OS scripts based on metrics M14 and M15?

The scripts underwent manual inspection and testing in a controlled environment, and the findings are included in the work archive. In general, each tool exhibited at least one error. The most common issues were related to incorrect usage of script parameters and inaccurate integer thresholds when generating CPU monitoring scripts

Table 6.15: Correctness metrics of generated shell scripts.

Language	GH Copilot		Tabnine		ChatGPT		CodeGeeX	
	M14	M15	M14	M15	M14	M15	M14	M15
Bash	9	0.9	7	0.7	7	0.7	8	0.8
PowerShell	8	0.8	7	0.7	9	0.9	5	0.5
All combined	17	0.85	14	0.7	16	0.8	13	0.65

GitHub Copilot generated the highest percentage of correct OS scripts (85%), while CodeGeeX produced the least (65%). As M15 is simply a ratio of M14 to the number of samples, scoring was based on a single metric, with GitHub Copilot earning 4 points, ChatGPT 3 points, Tabnine 2 points, and CodeGeex 1 point

Summary

In this summary section for Goal 1 experiments, we discuss the achieved overall quality of the generated source code and present the vulnerability analysis results. Additionally, we compare the relevant subset of results with those of related works. The summary concludes by evaluating the best tool in terms of source code quality based on the presented scoring system to meet Goal 1.

Examining the source code quality analysis by SonarCloud from a broad perspective, we observe that all ratings, except for maintainability, fall into the worst category. According to SonarCloud, it would take a total of 2 days and 4 hours to remediate bugs, 3 days and 7 hours for vulnerabilities, and 9 days and 3 hours for code smells.

SonarCloud has also identified 527 security hotspots, which require manual review. However, it is important to note that the presence of 527 security hotspots does not necessarily imply the existence of 527 vulnerabilities.

On the other hand, the debt ratio of the generated programs is only 0.8%, significantly below the A rating limit of 5%. Nevertheless, the results of the source code quality analysis indicate that the AI-generated code is not ready to be used out of the box, and further human work is needed to enhance the generated code.

In the OS shell scripting domain, the tools performed fairly well, producing 75% correct scripts. Out of the total 860 generated programs, 753 are valid (88%). When considering only interpreted languages, the tools generated 458 (99%) valid programs out of a possible 464. However, results for compiled languages are significantly worse, with the tools generating 295 (74%) valid programs out of 396. This is likely attributed to the more complex structure of projects in compiled languages, which generally require more configuration to be compilable. Consequently, it is not an easy task to achieve correctness on the first try with a single prompt.

A total of 255 vulnerabilities were detected by CodeQL, and 106 vulnerabilities were identified by SonarCloud. The vulnerabilities detected by CodeQL are evenly distributed across the three defined scenarios, with SecurityEval having 85 occurrences, CWE definition having 84, and CWE context 86. In contrast, SonarCloud results in-

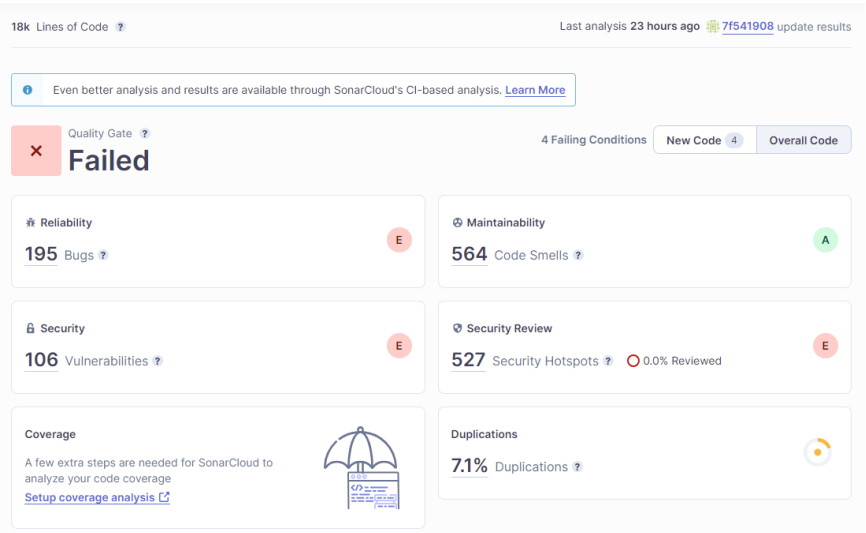


Figure 6.1: SonarCloud dashboard of the generated programs.

dicate variations between the scenarios, with SecurityEval having 46 occurrences, CWE definition having 29, and CWE context 31.

When comparing programming languages based on the number of vulnerabilities detected by CodeQL, Python had the highest count with 117 vulnerabilities, followed by JavaScript with 90. In contrast, C and C# exhibited lower vulnerability counts, with 28 and 20, respectively. SonarCloud identified the highest number of vulnerabilities in C (55) and C# (39), while Python had the least (12) and JavaScript none. Security hotspots were most prevalent in JavaScript (192), C (164), and Python (139), with C# having the least (39).

Related works have concentrated on detecting CWE vulnerabilities using CodeQL in Python source code generated by GitHub Copilot, ChatGPT, and other tools, reporting vulnerability rates of 40% [41], 34-43% [17], and 35.8% [43] in the generated programs. In this work, 40.2% of the generated Python programs are found to be vulnerable based on CodeQL analysis. This indicates stable characteristics of the generated code throughout the years, likely attributable to the fact that the distribution of vulnerabilities in the training data remains consistent or similar and the training methods have not focused on preventing vulnerabilities.

The most frequently detected vulnerability by CodeQL is *Flask app is run in debug mode*, occurring 36 times. According to the CWE classification, this vulnerability falls under CWE-215 Insertion of Sensitive Information Into Debugging Code and CWE-489 Active Debug Code. The second most frequent vulnerability, with 25 occurrences, is *Information exposure through an exception*, categorized under CWE-209 Generation of Error Message Containing Sensitive Information and CWE-497 Exposure of Sensitive System Information to an Unauthorized Control Sphere.

For SonarCloud, the most frequent vulnerability is shared by *CWE-120 Buffer Copy without Checking Size of Input* and *CWE-676 Use of Potentially Dangerous Function*, both occurring 44 times. The second most frequent vulnerability is *CWE-20 Improper Input Validation*, observed 25 times.

When the scores of tools are summed together for all research questions, it is clear that GitHub Copilot has dominated the scoring system, achieving the best metrics in five research questions (in one research question, the first placement was shared with other tools), as shown in Table 6.16. The second-best tool is CodeGeeX. This could be attributed to the fact that, since July 2023, CodeGeeX has utilized the second generation of the CodeGeeX model, which demonstrates much better performance in standard benchmarks than the first generation and other LLMs.

Tabnine secured the third position, while ChatGPT emerged as the least performing tool among the selected ones, which might be surprising considering its widespread popularity. However, it's essential to note that ChatGPT is not exclusively designed for code generation; it serves as a general chatbot. Another factor contributing to this result could be the utilization of ChatGPT 3.5 instead of the latest version, ChatGPT 4. The reasons behind this decision are detailed in the Threats to Validity section.

Table 6.16: Resulting scores of the tools across all research questions in the primary goal. The best-performing tools in each research question are highlighted in blue.

	GH Copilot	Tabnine	ChatGPT	CodeGeeX
RQ1	18	**22**	14	18
RQ2	**21**	8	17	16
RQ3	17	16	10	**23**
RQ4	**22**	18	15	18
RQ5	**6**	**6**	3	**6**
RQ6	3	2	**4**	1
RQ7	**16**	12	9	13
RQ8	**4**	2	3	1
Final Score	**107**	86	75	96

6.2 Goal 2

The Tabby framework [50] is utilized for developing the custom code generation tool. The main steps of the process involve running a Tabby server in a Docker container and then configuring the Tabby IDE plugin to connect to the URL of the Tabby server. The server can be initiated locally using an official Docker image from Tabby:

```
docker run -it --gpus all -p 8080:8080 -v $HOME/.tabby:/data \
    tabbyml/tabby serve --model TabbyML/StarCoder-1B --device cuda
```

Tabby offers several open-source LLMs in its model registry. In the command above, the StarCoder with 1B parameters is used. The LLM can use a GPU ('–device cuda' in the Docker command) or a CPU ('–device cpu') for its computations. Once the server is up and running, it defines three main endpoints:

- /v1/completions
- /v1/events
- /v1/health

The health endpoint is used to determine the status of the server, and the completions endpoint is where the IDE plugin sends requests for code generation.

For both Visual Studio Code and Intellij IDEs, the Tabby plugin can be installed from their marketplaces. If using the default configuration, the Tabby server runs on port 8080 [50]. In this case, the URL of the Tabby server configured in the plugin settings would be http://localhost:8080.

Cloud Deployment

Since LLMs require extensive computational resources, local deployment is often not feasible. Fortunately, the Tabby server can be easily run in a Docker container, making deployment to cloud environments straightforward. The Tabby documentation provides step-by-step instructions for deploying the solution to a serverless cloud platform Modal[5]. The serverless on-demand deployment model charges only for the used RAM and CPU/GPU time.

To deploy a Tabby server using Modal, Python 3.7 or later is required. The initial step involves installing the Python modal library

```
pip install modal
```

Authentication to Modal requires an API token, which can be obtained using the Python Modal library:

```
model token new
```

After completing the Modal setup, the next step involves using a deployment script. Tabby conveniently provides a suitable default script. This script utilizes the Nvidia T4 GPU, the cheapest GPU option offered by Modal, and is sufficient for the performance requirements of StarCoder 1B LLM.

To execute the deployment script locally, run:

```
model server deployment_script.py
```

The script defines the server's URL as *https://<USERNAME>–tabby-server-starcoder-1b-app-dev.modal.run*. To verify that the server is running, the health check endpoint can be used:

```
curl --location \
'https://<username>--tabby-server-starcoder-1b-app-dev.modal.run/v1/health'
```

5. https://modal.com/

The final step is to configure the IDE plugin to use the Tabby server deployed on Modal. This can be achieved in the 'API: Endpoint' settings of the plugin in Visual Studio Code.

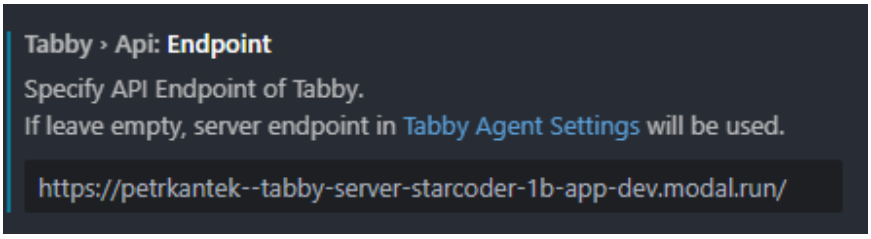


Figure 6.2: Tabby plugin configuration in Visual Studio Code.

For IntelliJ IDE, the process remains identical. Install the Tabby IDE plugin from the Jetbrains marketplace and configure it to send requests to the Tabby server URL hosted on Modal. Once the configuration for either IDE is successfully completed, the requests from the IDE to the Tabby server can be monitored using Modal's app dashboard.

Figure 6.3: Monitoring metrics of the Tabby server on Modal.

6.3 Threats to Validity

This work carries several threats to the validity of the results. A threat to the *internal validity* arises from the nondeterminism of LLM outputs. The same prompt can yield different results, potentially preventing complete reproducibility of the data set. However, to maintain uniformity in the data generation process, the AI tools were employed consistently with identical prompt structures and comments.

Another threat to internal validity is the manual inspection of research question 8. To address this concern, a list of detected errors has been made available in the work's archive. This list can be used for comparison with the generated scripts, offering a means to mitigate the potential impact of this threat.

A threat to the *external validity* stems from the choice of ChatGPT version (3.5) rather than the latest version (4). This decision was made because OpenAI placed new users of ChatGPT with GPT-4 on a wait list, rendering the newest model inaccessible during the development of this work.

The selection of specific static analysis tools could impact the experimental results and the determination of the best AI tool in the primary goal. To mitigate this threat, a set of well-known, production-ready, and battle-tested tools was deliberately chosen. Additionally, the selection of analysis tools took into account those used in related works to facilitate a more meaningful comparison of results.

Sampling bias introduces an external validity threat by generating programs based on CWE scenarios susceptible to vulnerabilities. Furthermore, the generated code tends to be relatively short, with the longest example consisting of around 200 lines of code. Utilizing such a data set may not fully simulate real-world projects. However, it is worth noting that other data sets, such as SecurityEval [44], FormAI dataset [45], or HumanEVal [13], also contain smaller programs arranged in specific scenarios.

7 Conclusion

This work introduced deep learning models for code generation, including the essential Transformer architecture and LLMs built around the concept of attention mechanisms. Several state-of-the-art LLMs were described, establishing the theoretical foundations. The following chapter introduced popular AI-based code generation tools, such as GitHub Copilot, ChatGPT, and CodeGeex, and explored how they can be integrated into IDEs. The work also delved into the concepts of software and source code quality, along with their associated metrics. A significant focus was placed on vulnerabilities, including the CWE Top 25 list, CodeQL, and SonarCloud.

As part of the experiments, a multilingual data set consisting of 860 autonomous programs with approximately 18,000 lines of code was collected. This data set includes programs written in Python, C, C#, JavaScript, Bash, and PowerShell. The programs are based on the SecurityEval dataset[44], CWE definitions, and GitHub CodeQL documentation.

The experiments were designed to achieve two primary goals, address eight research questions, and evaluate 15 metrics. The main objective was to identify the best AI-based code generation tool through a scoring system that considered values from various metrics. GitHub Copilot emerged as the top-performing tool, scoring 107 points, surpassing the second-best tool CodeGeex, which accumulated 96 points.

Each research question provided an insight into AI-driven source code generation. For example, the tools, on average, generated 40.2% of Python programs containing at least one vulnerability. This finding aligns with results reported in related works [41, 42, 43]. It is evident that no significant, game-changing modifications have been made to the tools and the underlying LLMs since the related works were published that would significantly alter their performance in either direction.

Future Work

Future work can take diverse paths, and one promising involves further expanding and curating the data set of generated programs, ulti-

mately releasing it in a meaningful format. This expansion could create a more extensive range of scenarios and use cases, providing valuable insights into the performance and adaptability of code generation models.

In pursuit of the secondary objective, a custom LLM for code generation was deployed as a serverless application. This deployment opens the door to further possibilities by considering the fine-tuning of the LLM on security datas ets or other specialized data sets. This targeted approach aims to enhance the model's proficiency in generating code that adheres to high-quality standards, particularly in the area of source code quality metrics. Such refinements could potentially contribute to advancements in code quality and security in software development practices.

Another direction could be in the field of self-healing applications. Given that AI models can effectively generate code, fine-tuned instruction models could be employed to automatically fix errors and bugs in applications by prompting LLMs with the particular error and an instruction to fix it[1].

1. https://stackoverflow.blog/2023/06/07/self-healing-code-is-the-future-of-software-development/

Bibliography

1. BASILI, Victor R.; WEISS, David M. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*. 1984, vol. SE-10, no. 6, pp. 728–738. Available from DOI: 10.1109/TSE.1984.5010301.
2. HINDLE, Abram et al. On the naturalness of software. In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 837–847. Available from DOI: 10.1109/ICSE.2012.6227135.
3. GABEL, Mark; SU, Zhendong. A Study of the Uniqueness of Source Code. In: Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 147–156. FSE '10. ISBN 9781605587912. Available from DOI: 10.1145/1882291.1882315.
4. VASWANI, Ashish et al. Attention is all you need. *Advances in neural information processing systems*. 2017, vol. 30.
5. ZHAO, Wayne Xin et al. A Survey of Large Language Models. *arXiv preprint arXiv:2303.18223*. 2023. Available also from: http://arxiv.org/abs/2303.18223.
6. TOUVRON, Hugo; MARTIN, Louis; AL., Kevin Stone Et. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. Available from arXiv: 2307.09288 [cs.CL].
7. TOUVRON, Hugo et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*. 2023.
8. KAPLAN, Jared et al. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*. 2020.
9. BROWN, Tom et al. Language models are few-shot learners. *Advances in neural information processing systems*. 2020, vol. 33, pp. 1877–1901.
10. OPENAI. GPT-4 Technical Report. *ArXiv*. 2023, vol. abs/2303.08774. Available also from: https://api.semanticscholar.org/CorpusID:257532815.
11. RADFORD, Alec; NARASIMHAN, Karthik; SALIMANS, Tim; SUTSKEVER, Ilya, et al. Improving language understanding by generative pre-training. 2018.
12. RADFORD, Alec et al. Language models are unsupervised multitask learners. *OpenAI blog*. 2019, vol. 1, no. 8, p. 9.

13. CHEN, Mark et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*. 2021.
14. ROZIÈRE, Baptiste et al. *Code Llama: Open Foundation Models for Code*. 2023. Available from arXiv: 2308.12950 [cs.CL].
15. FACE, Hugging. *Hugging Face* [https://huggingface.co]. 2023. Accessed: December 1, 2023.
16. FACE, Hugging. *CodeParrot* [https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot]. 2021.
17. LI, Raymond et al. *StarCoder: may the source be with you!* 2023. Available from arXiv: 2305.06161 [cs.CL].
18. PENEDO, Guilherme et al. *The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only*. 2023. Available from arXiv: 2306.01116 [cs.CL].
19. HOFFMANN, Jordan et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*. 2022.
20. LI, Yujia et al. Competition-level code generation with alphacode. *Science*. 2022, vol. 378, no. 6624, pp. 1092–1097.
21. GITHUB. *The world's most widely adopted AI developer tool.* 2023. Available also from: https://github.com/features/copilot. Accessed: December 1, 2023.
22. BLOG, GitHub. *Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*. 2022. Available also from: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/. Accessed: December 1, 2023.
23. OPENAI. 2023. Available also from: https://openai.com. Accessed: December 1, 2023.
24. TABNINE. 2023. Available also from: https://www.tabnine.com. Accessed: December 1, 2023.
25. CODEGEEX. 2023. Available also from: https://codegeex.cn/en-US. Accessed: December 1, 2023.
26. CODEGEEX. 2023. Available also from: https://codegeex.cn/en-US. Accessed: December 1, 2023.
27. CODEIUM. 2023. Available also from: https://codeium.com. Accessed: December 1, 2023.
28. AI, Blackbox. 2023. Available also from: https://www.blackbox.ai. Accessed: December 1, 2023.

29. CURSOR. 2023. Available also from: https://cursor.sh. Accessed: December 1, 2023.
30. ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.
31. ISO/IEC 25010. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011.
32. CORPORATION, The MITRE. *2023 CWE Top 25 Most Dangerous Software Weaknesses*. 2023. Available also from: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. Accessed: November 24, 2023.
33. FOUNDATION, The OWASP. *OWASP Top 10 2021*. 2021. Available also from: https://owasp.org/Top10. Accessed: November 23, 2023.
34. AVGUSTINOV, Pavel; DE MOOR, Oege; JONES, Michael Peyton; SCHÄFER, Max. QL: Object-oriented queries on relational data. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
35. SYNOPSYS. *Linux Kernel*. 2023. Available also from: https://openhub.net/p/linux/analyses/latest/languages_summary. Accessed: November 23, 2023.
36. MCCABE, T.J. A Complexity Measure. *IEEE Transactions on Software Engineering*. 1976, vol. SE-2, no. 4, pp. 308–320. Available from DOI: 10.1109/TSE.1976.233837.
37. HALSTEAD, Maurice H. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.
38. COLEMAN, D.; OMAN, P.; ASH, D.; LOWTHER, B. Using Metrics to Evaluate Software System Maintainability. *Computer*. 1994, vol. 27, no. 08, pp. 44–49. ISSN 1558-0814. Available from DOI: 10.1109/2.303623.
39. LETOUZEY, Jean-Louis. The SQALE Method for Managing Technical Debt. 2016.
40. CHIDAMBER, S.R.; KEMERER, C.F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 1994, vol. 20, no. 6, pp. 476–493. Available from DOI: 10.1109/32.295895.

41. PEARCE, Hammond et al. Asleep at the keyboard? assessing the security of github copilot's code contributions. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.
42. ASARE, Owura; NAGAPPAN, Meiyappan; ASOKAN, N. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*. 2023, vol. 28, no. 6, pp. 1–24.
43. FU, Yujia et al. Security Weaknesses of Copilot Generated Code in GitHub. *arXiv preprint arXiv:2310.02059*. 2023.
44. SIDDIQ, Mohammed Latif; SANTOS, Joanna C. S. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In: *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 29–33. ISBN 9781450394574. Available from DOI: 10.1145/3549035.3561184.
45. TIHANYI, Norbert et al. The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. In: *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. <conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 33–43. PROMISE 2023. ISBN 9798400703751. Available from DOI: 10.1145/3617555.3617874.
46. NIU, Liang; MIRZA, Shujaat; MARADNI, Zayd; PÖPPER, Christina. CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023, pp. 2133–2150. ISBN 978-1-939133-37-3. Available also from: https://www.usenix.org/conference/usenixsecurity23/presentation/niu.
47. HUANG, Yizhan et al. Do not give away my secrets: Uncovering the privacy issue of neural code completion tools. *arXiv preprint arXiv:2309.07639*. 2023.
48. NGUYEN, Nhan; NADI, Sarah. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1–5. MSR

'22. ISBN 9781450393034. Available from DOI: 10.1145/3524842.3528470.

49. WHITE, Jules et al. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. Available from arXiv: 2302.11382 [cs.SE].

50. TABBYML. *Tabby* [https://github.com/TabbyML/tabby]. 2023. Accessed: December 10, 2023.

A Attached Source Code

This supplementary material contains a description of the attached source code files.

```
root
├── analysis → Python helper scripts for the experiments
├── llm_deployment → Python scripts for deployment of an LLM
├── results → folder with Excel files containing experimental
│   results
├── templates → folder with template files for generating
│   programs
├── vulnerability_analysis → the data set with generated programs
└── README.md → file with other detailed information
```