PANTHÉON SORBONNE

EVALUATION METHODS FOR GENERATED CODE

CURRENT AND FUTURE METHODS TO EVALUATE THE QUALITY OF GENERATED CODE

AUTHORS

ETIENNE BAUMGARTNER

Université Panthéon-Sorbonne

MSc MIAGE

Mémoire

PARIS, MARCH 2024

PANTHÉON SORBONNE

EVALUATION METHODS FOR GENERATED CODE

CURRENT AND FUTURE METHODS TO EVALUATE THE QUALITY OF GENERATED CODE

AUTHORS

ETIENNE BAUMGARTNER

Student No. 12309480

SUPERVISOR

NICOLAS HERBAUT, ASSOCIATE PROFESSOR

PARIS, MARCH 2024

Declaration of Authorship

I declare on my honour that the work presented in this dissertation, entitled "Evaluation methods for generated code," is original and was carried out by Etienne Baumgartner (12309480), under the supervision of Professor Nicolas Herbaut, Associate professor (nicolas.herbaut@univ-paris1.fr).

Paris, March 2024

[illegible]

Etienne Baumgartner

Acknowledgements

I would like to extend my gratitude to my former boss, Pascal Müller. During one of our discussions regarding the influence of AI, he shared a personal perspective. He remarked that it is the first time in this digital era, that he perceives a threat to the job as software engineer — a sentiment that has remained with me ever since.

His words prompted me to ponder a fundamental question: What distinguishes generated code from what a software engineer such as myself produces? This inquiry served as the catalyst for formulating my research questions, guiding the trajectory of my investigation.

Abstract

Recent advancements in large language models have led to significant progress in the code generation tasks. However, the evaluation of these models has primarily relied on metrics intended to assess natural language, only recently shifting towards measuring functional correctness using unit testing. As code generation models are able to operate in larger problem domains and wider scopes, the current evaluation setup needs reevaluation.
This thesis illustrates currently used evaluation methods and their origins and researches the potential integration of software engineering metrics into the evaluation process. The study identifies several widely accepted software metrics suitable for indicating the quality of generated code.

Keywords: Evaluation metrics, Software metrics, Code quality, Code generation, Functional correctness, Quality indicators

Contents

List of Figures

List of Tables

Acronyms

ASG	Abstract Semantic Graph. *(p. 18, 20)*
AST	Abstract Syntax Tree. *(p. 18–20, 29)*
CBO	Coupling between Objects. *(p. 29, 30, 33)*
CC	Cyclomatic Complexity. *(p. 28, 29, 33)*
DIP	Dependency Inversion Principle. *(p. 26, 27)*
DIT	Depth of Inheritance Tree. *(p. 30, 33)*
FI	Fan-In. *(p. 32, 33)*
FO	Fan-Out. *(p. 32, 33)*
GPT	Generative Pre-trained Transformer. *(p. 2)*
ISP	Interface Segregation Principle. *(p. 26)*
LLM	Large Language Model. *(p. 3–6, 22, 39)*
LLOC	Logical Lines of Code. *(p. 31)*
LOC	Lines of Code. *(p. 31, 33, 34)*
LOCM	Lack of Cohesion in Methods. *(p. 30, 33)*
LSP	Liskov Substitution Principle. *(p. 26)*
MBPP	Mostly Basic Programming Problems. *(p. 21, 22)*
NLP	Natural Language Processing. *(p. 2, 3, 7, 13, 15, 18, 20, 22, 35, 37, 39)*
NOA	Number of Attributes. *(p. 32, 33)*
NOC	Number of Children. *(p. 31–33)*
NOM	Number of Methods. *(p. 32, 33)*
OCP	Open-Closed Principle. *(p. 25)*
OOP	Object-Oriented Programming. *(p. 27, 28, 36)*
PLOC	Physical Lines of Code. *(p. 31, 33)*

RFC	Response for a Class. *(p. 31, 33)*
RNN	Recurrent Neural Network. *(p. 7, 8)*
SRP	Single Responsibility Principle. *(p. 25)*
WMC	Weighted Methods per Class. *(p. 29, 33)*

1 Introduction

Natural Language Processing (NLP) models encompass a set of machine learning algorithms that exploit the fundamental structure of human language. These models leverage the statistical distribution inherent in human language, effectively predicting word sequences and generating extensive, syntactically coherent text. Interestingly, when trained on code, such models demonstrate the intrinsic ability to generate functional code. Barr et al. (2016) and Allamanis et al. (2018) have previously formulated this exact observation after researching the distribution of code and its similarities with natural language. Both domains are heavily influenced by reoccurring patterns and structures that can be captured by statistical models - a concept encapsulated in the *naturalness hypothesis* for code.

Revolutionary breakthroughs in NLP such as the development of the transformer by Vaswani et al. (2017) and the subsequent release of Generative Pre-trained Transformer (GPT) based models simutlanously opened up new avenues for the code generation task. Notably, (M. Chen et al., 2021) published Codex, a model tailored specifically for the coding task and precursor to GitHub's Copilot. Leveraging the naturalness of code, Codex is a GPT-3 model, originally designed for natural language processing, that has been fine tuned for the coding task.
Following the emergence of new models designed specifically for the code generation task, comes the demand for suitable evaluation metrics. Allamanis et al. (2018) highlights during their formulation of the naturalness hypothesis that there is a lack of well-suited and universally accepted evaluation metrics for the code generation task. An issue that continues to be a subject of ongoing research. Until recently, models created for coding tasks were still evaluated with methods designed for NLP. An unhealthy evaluation practise that leads to the prioritization of linguistic capabilities over coding proficiency. Code possesses distinct properties that warrant additional and separate evaluation, despite being inherently connected to natural language through the naturalness hypothesis.

(M. Chen et al., 2021) recognized this necessity for a code-specific evaluation by introducing a metric called HumanEval. HumandEval solely focuses on assessing functional correctness by unit testing. Presently, code generation models primarily operate within a narrow scope, focusing on functions and statements. For this use case, evaluation based on NLP capabilities and functional correctness may suffice. However, as future models advance their understanding and develop capabilities operate in class or project level scopes, the current evaluation methods and their underlying metrics needs to be reconsidered. Pursuing the notion of semantic, syntactical, and functional correctness is essential, but it will not suffice to evaluate all the properties that code should possess. Programs of this scale demand methods applied during the quality assessment of large-scale software. It should be considered that future models would be evaluated on the basis of quality indicators found in traditional software engineering.

This thesis answers the more general research question: *"How do we measure the quality of generated code"*. Furthermore it explores the possibilities of enhancing the current metrics standards. The focus lies in finding potential metrics applied in quality assurances of traditional software systems that could be used for the evaluation of LLMs. As such, this thesis develops an answer to a subsequent research question: *"Can traditional software engineering metrics be used to assess the quality of generated code?"*

2 Background

This section provides insights into the and their underlying theory of LLMs.

2.1 Code Naturalness

Statistical properties in natural language are leveraged in a diverse set of application ranging from translation to error correction. Recent LLMs demonstrate what models based on the statistical distributions of language can achieve in various contexts. Among those numerous contexts, the generation of code plays an interesting role. Models that are not specifically trained on code show capabilities of assisting and reasoning in various code generation tasks solely based on the presence of code in their training set. The question is where does this affinity of language models to naturally generate code come from. Barr et al. (2016) , in their paper titled *"Naturalness of Software"*, suggests that various statistical properties observed in natural language similarly apply to code. Consequently, it can be rightfully assumed as it has been shown by M. Chen et al. (2021) , that LLMs are capable of achieving proficient performance in code generation tasks, given appropriate fine-tuning and training.

The idea of using the actual distribution of natural language corpora is the reason for the abilities observed in today's large language models. The shift from the typical top-down approach where a method follows grammatical and syntactic rules towards the more data-driven approach has been successfully applied to use cases such as language translations. This success dates back many years to the 1980 (Jones, 1994). Despite a potential for complexity, human communication tends to be simplistic on average. This simplicity arises from the repetitive usage of terms and phrases, implying that a basic set of vocabulary is sufficient for the transfer of information. In the context of a normal conversation, it could be argued that it is preferable to use easier-to-understand phrases and terms. Such arguments find their analogy in the domain of software engineering, where simple-to-understand and readable code holds greater value than complex and lengthy code. In that way, code shows many parallels with natural language. Barr et al. (2016) and later Allamanis et al., 2018 have shown that code possesses the same repetitive and recurring nature as natural language does. Hence, it is also susceptible to the potential implications its probabilistic properties hold. Repetitiveness and simplicity in the context of software engineering is even more desirable since it enhances reusability, readability and maintainability.

Making the connection between code and natural language has already been proposed by Prof. Knuth (Knuth, 1984). He suggests considering programming code as a kind of literature and introduces the concept of writing code akin to human-to-human communication.

> *"[...] considering programs to be works of literature. Hence, my title: "Literate Programming." Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. The practitioner of literate programming can be regarded as an essayist, [...]"*

Barr et al. (2016) rephrase this idea by introducing their hypothesis on *"Code naturalness"*:

> *"Programming languages, 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."*

(Allamanis et al., 2018) simplify even further:

> *"Software is a form of human communication. Software corpora have similar statistical properties to natural language corpora and these properties can be exploited to build better software engineering tools."*

2.1.1 Code vs. Text

While exhibiting numerous parallels, code demonstrates significant disparities to natural language. These distinctions have ramifications for both the generated code and the underlying models used for the generation process. Recognizing these distinctions is essential as they will impact the training and the evaluation. The following paragraphs explain these differences closer.

At this juncture, the term "token" is defined, as it is frequently employed in the following sections.

Token:
In the context of the LLM tokens present the basic units of read input and generated output.
For natural language, tokens can be words, punctuation, etc. whereas for natural code tokens can be function identifiers, keywords, operators, literals as well as punctuation.

Neologism Natural code is suspect to a lot of neologism, for example in function identifiers. But in comparison with natural language it holds a considerably smaller set of tokens. On one hand, neologism can impact the evaluation of a model. Generated code normally differs syntactically from a reference solution but does not so semantically. On the other hand, it can be argued that the presence of neologisms promotes the significance of often encountered keywords like function, class, public, etc. This changes the way the model generates a solution.

Versioning Code and their programming languages are updated punctually. The differences in language versions can lead to potential problems while prompts are interpreted and code is generated. Certain functionalities might only be provided by a certain language version. In contrast, natural language changes dynamically and gradually. Depending on the problem domain a model needs to facilitate the newest functionality or understand legacy assignments. With time such new versions and functionality can be integrated into future iterations of a model.

Execution Code is executable and carries potential real-life implications which can pose risks. The biggest risk arises in automatic execution of generated code, as it is needed in certain evaluation frameworks, such as functional correctness trough unit testing. Similarly to the training corpora of natural language, it is to assume that code corpora can contain malicious examples which make the model theoretically capable of generating malicious code. It is crucial to be aware of this fact while creating evaluation frameworks. Systems that execute these tests must acknowledge the threat by implement preemptive actions such as ensuring the execution in a secure environment.

Semantic robustness Code lacks the semantic robustness of natural language. Robustness is less significant in natural language as minor alterations of mistakes normally do not overly alter the semantic meaning. In the domain of programming, such alterations could drastically affect the functionality of the program.

Modern LLMs focusing on the task of code generation must consider these points by having appropriately selected training data, as well as meaningful evaluation frameworks that can account for semantic robustness, malicious code, and neologisms.

2.2 Models

In this section, we delve into the evolution of models that have been and are used for code generation, from simple n-grams to the transformer introduced in 2017. Focus lies on the importance of understanding the capabilities of newer generations, and how they differ to previous versions.

The second part of this sections present some data for the widely successful transformer models, illustrating the progress that has been achieved in a short amount of time.

2.2.1 Model basis

Most models use the probabilistic distribution of code to generate a sequence of tokens. Sequences are created by merging tokens while calculating the probability of the current token based on previously occurring tokens.

$$P(t_n|t_1, \dots, t_{n-1}) \tag{2.1}$$

N-gram One of the well-known specific token-based models is the n-gram model, which finds its particular use as a language model and is the basis of a large number of evaluation metrics for the NLP. N-grams are able to find statistical dependencies in sequences quite easily by using the probabilistic distribution noted above 2.1 for each token in the sequence. Such a model can generate text and code by following the observations made on patterns in a training set. They are simple and powerful but tend to fall behind other techniques in regard to long-ranged dependencies, meaning they will have difficulties connecting information from the end of a prompt with the beginning.

RNN Recurrent Neural Network (RNN)s are also models used for token generation. These models are not as context-restricted as n-grams tend to be. RNNs can better capture long-range dependencies in their hidden layers and are therefore well equipped to outperform the more localized scope of n-grams. On the other hand, the construction of RNN is more involved and demands a larger amount of data and training time.

Encoder / Decoder Cho et al. (2014) introduced the encoder-decoder architecture to improve the performance of statistical machine translation such as the translation from English to French and vice versa. The architecture proposes the use of two RNNs, one acting as an encoder, mapping variable-length token sequences to a fixed-length intermediate vector representation, while the second RNN decoding the vector representation back to a variable-length sequence. Encoder-decoder models can capture word and phrase regularities on syntactic and semantic levels.

Transformer Vaswani et al. (2017) criticizes the long-range dependency problem in encoder-decoder models. Models suffering from long-range dependency problems are not well suited to connect input information with a large positional distance between them. As such this feat is hard to achieve with RNNs as shown in Kolen et al. (2001) . Attention algorithms have been introduced to address the long-range dependency problem. This method adds an attention-based representation to input sequences capturing the relations of key tokens and their importance in the sequence. This method has been applied successfully to encoder-decoder models. Transformer models solely use attention mechanisms and do not rely on any recurring or convolutional layers to capture the dependency of input and output sequences. As such they are faster and more efficient and in the translation case mentioned in Vaswani et al. (2017) and they outperformed other models based on RNNs.
The recent models that have garnered a lot of attention are based on exactly that attention based transformer method.

2.2.2 Model sizes

Models have been growing in capability and size over the recent years. This means newer models requires more financial and computational resources than previous ones. There is a stark gap between closed and closed-sourced models, with the performance of closed-sourced models clearly in the lead and open-sourced models playing catch up. Two major companies, Google and OpenAI, are leading the closed-source models, as illustrated in the following figures. While the best-performing models are generally not freely accessible, there has been some effort from open-source communities and enterprises like Meta to create transparent and pre-trained models. Especially noteworthy are collaborative initiatives from multiple researchers such as the creation of BLOOM (Workshop et al., 2022) in 2022.

To demonstrate the general growth of the model sizes this paper explains size indicators and presents graphical illustration showing different models and their sizes.

Parameter count indicates the capability of learning and expressing more complex relations on data. It is not implicitly true that higher numbers in parameters count will lead to a better performing model. But the potential and adeptness of such models are generally higher. Size and cost optimization allow for smaller and smaller models that can outperform their larger predecessors. It is crucial to note that multiple versions with different parameter counts of the same model exist. Different sizes are intended for different use cases.

Context window length indicates the number of tokens a model can take as input. It describes the size of the input/output sequence a model can operate on. In general context length indicates the information and dependency a model can operate on. While a larger context size can be beneficial and produce more sophisticated answers it is also directly connected to a higher cost in training.

A series of models and their parameter count and token window size is illustrate in Table 2.1 and in shown in Figure 2.1 and Figure 2.2.

Table 2.1: *Parameter count and token window size per model and year.*

Year	Producer	Model	Parameter 10^9	Token Window 10^3
2018	Google	BERT [1] Devlin et al. (2019)	0.34B	-
2019	OpenAI	GPT-2[2] Alec et al. (2019)	1.5B	1K
2020	OpenAI	GPT-3 Brown et al. (2020) *OpenAI Platform Documentation* (n.d.)	175B	16K
2022	Google	PalmE Driess et al. (2023)	540B	-
2022	BigScience	BLOOM Workshop et al. (2022)	176B	-
2022	Facebook	LAMA Touvron et al. (2023b)	65B	2K-4K
2023	Facebook	LAMA2 Touvron et al. (2023a) S. Chen et al. (2023)	70B	32K
2023	Mistral AI	Mixtral 8x7B Jiang et al. (2024)	12.9B	32K
2023	OpenAI	GPT-4 *OpenAI Platform Documentation* (n.d.)	-	128K
2024	Google	Gemini *Google AI Blog* (n.d.)	-	128K - 1000K

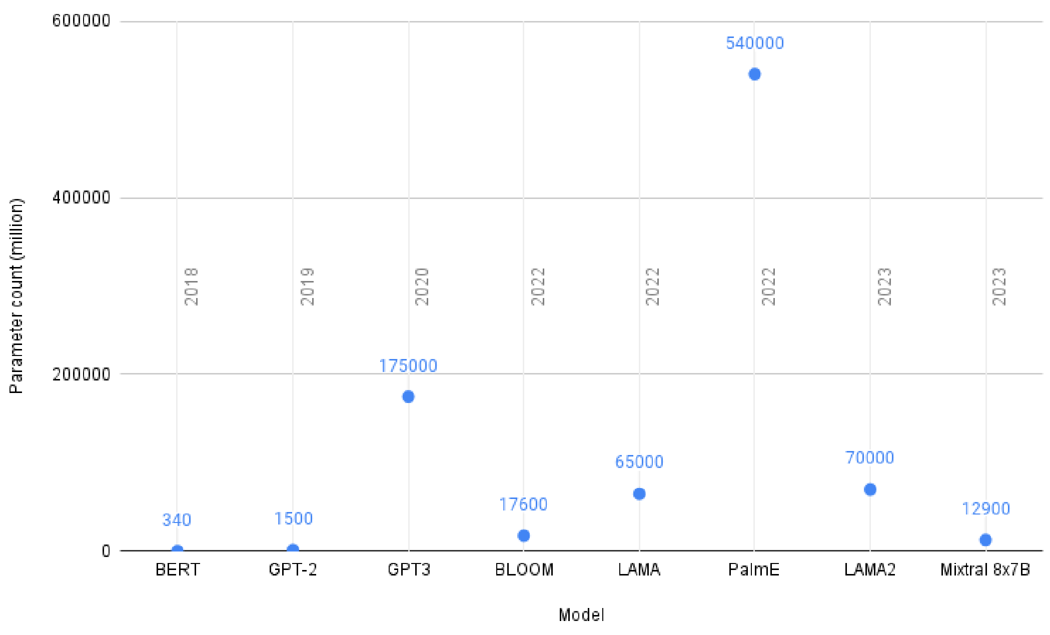

Figure 2.1: *Parameter count of various open and closed source models*

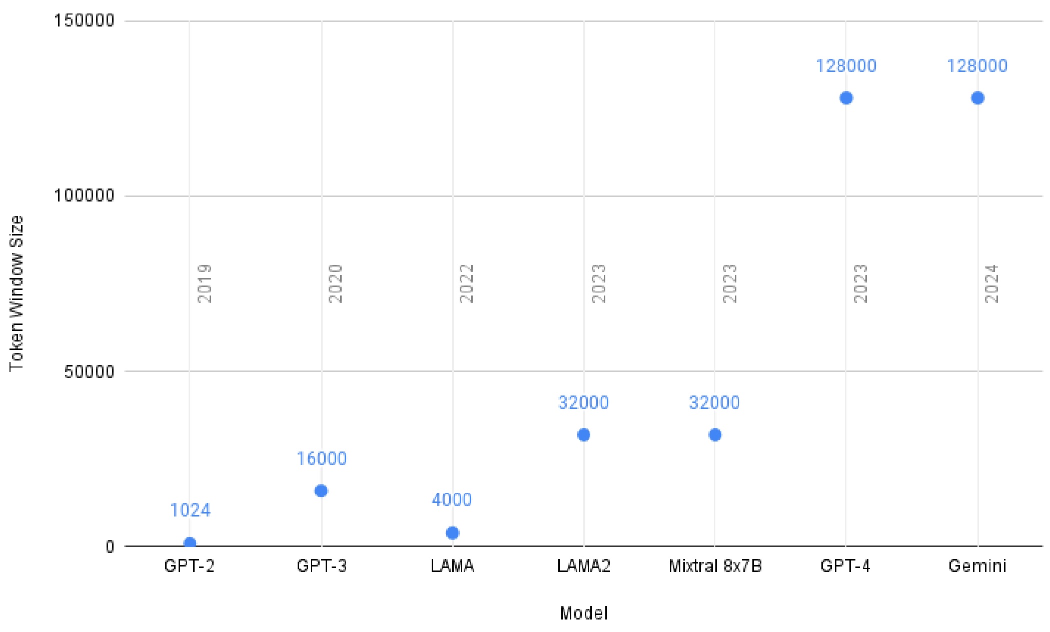

Figure 2.2: *Token window size for various open and closed source models*

3

Methodology

This thesis answers the following to research questions:

How do we measure the quality of generated code
Can traditional software engineering metrics be used to assess the quality of generated code?

This section describes the applied methodology to find relevant research. Several techniques were used.

1. Search by string
2. Snowballing
3. Direct search

Search string

The general research topic for this thesis can be summarized in the question: How do we *measure* the *quality* of *generated code*? To find relevant connected papers a search based on keywords has been initiated. Several search iteration with different variations of keywords from the research topic has been used to find suitable articles on the MIAGE scholar platform (*Scholar MIAGE* n.d.). The final search string resulting in 99 papers is presented below [1].

[1] Moment of writing: March 2023

```
(TITLE-ABS-KEY("quality") OR TITLE-ABS-KEY("score") OR
TITLE-ABS-KEY("benchmark") OR TITLE-ABS-KEY("metrics"))
AND
(TITLE-ABS-KEY("evaluation") OR TITLE-ABS-KEY("measurement") OR
TITLE-ABS-KEY("classification"))
AND
(TITLE-ABS-KEY("code generation"))
AND
(SUBJAREA("COMP"))
AND
PUBYEAR > 2020
```

The first three parts of the query represents logical OR keywords. The first is a variation of the key word *quality*. The second part follows the same structure for the key word *measure*, while the third key word *generated code* is precise enough not to necessitate further variations. The three query parts are combined with a logical AND describing that at least one key word of each part needs to be present in either title, abstract or keyword of any potential result. Finally the query is restricted by the research area of computer science and publications dates are limited to at least 2020.

Further selection on the initial set has been done by inspecting titles and abstracts as well as keywords of the papers. Additionally, only freely accessible articles have been considered initially.

Snowballing

Reading the first few articles were inspiring and initiated the reevaluation of the current selection. It was clear that most of the articles based their findings on previously established results by other authors. As such with the snowballing technique of reference searching the initial papers were found. This selection would built the foundation of this thesis.

Reference search was a fortuitous endeavor, since it showed that the second research question this paper aims to answer only found little to none attention in the research community so far.

Direct search

The second question of this thesis is related to traditional software metrics. No keywords have been included in the search string concerning this particular topic nor was any research found that was directly working on the same area. As such, relevant papers in the domain of software metrics have been directly search.

Putting it together

Finally these three methods resulted in the articles that build the basis for this thesis. Search strings included papers discussing automated Natural Language Processing (NLP) metrics as well as established functional correctness techniques. Snowballing on these papers revealed the concept of Code naturalness and the additional direct search helped integrate well known traditional software metrics.

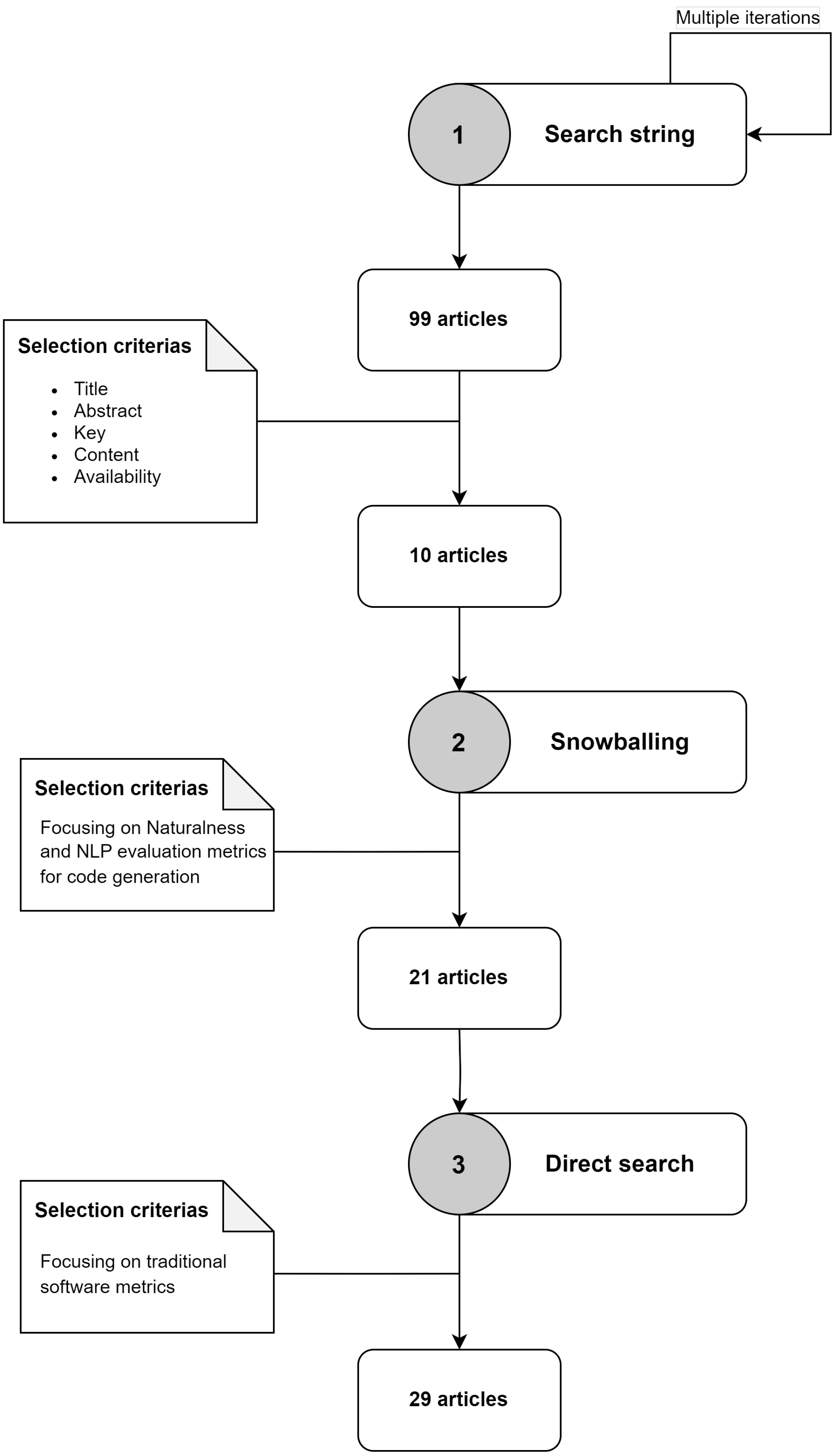

Figure 3.1: *Research methodology*

4 Results

In the results or this thesis the various methods currently used in evaluating code generation models are presented. Evaluation is essential in research, as it provides the means to understand the capabilities of existing models and clearly identifies areas requiring further improvements. As such, Section 4.1 answers the research question *"How do we measure the quality of generated code?"* whereas Section 4.2 answers the second research question *"Can traditional software engineering metrics be used to asses the quality of generated code?"*.

At this point concepts concerning the evaluation task for code generation are defined.

Evaluation set
An evaluation set is the set of task that is used to measure the performance of a model

Evaluation metric
An evaluation metric is the quantifiable value used to assess the performance of a model. It can be described as a quantifier that compares a generated solution against the reference solution of a given task.

4.1 Existing evaluation

The following sections present existing evaluation methods and gives more context how they asses code. An overview of those metrics can be found in Table 4.1

4.1.1 NLP Methods

BLEU (Bilingual Evaluation Understudy)

Interestingly enough, for the comparison of code generation models NLP methods such as BLEU (Papineni et al., 2002) are used. BLEU is intended to evaluate machine translation from one natural language to another by using metrics that use calculation over n-grams for generated and reference solutions. The BLEU method is described by the following: *"The closer a machine translation is to a professional human translation, the better it is"*.

BLEU uses a numerical metric based on n-gram precision to calculate a score that indicates closeness of a generated solution to a reference solution.

$$p_n = \frac{|S^n_{\text{ref}} \cap S^n_{\text{gen}}|}{|S^n_{\text{ref}}|} \tag{4.1}$$

Where

$$S^n_{\text{ref}} = \text{set of n-grams of the reference solution}$$
$$S^n_{\text{gen}} = \text{set of n-grams of the generated solution}$$

In other words, the p_n is calculated by dividing the total number of overlapping n-grams between the generated solution and the reference solution by the total number of n-grams in the reference solution.

Additionally a Brevity penalty factor is used to account for the penalty of short generated solutions. If their length is significantly smaller than its reference solution, their score is also suffering.

$$\text{BP} = \begin{cases} 1 & \text{g > r} \\ e^{(1-\frac{\text{r}}{\text{g}})} & \text{otherwise} \end{cases} \tag{4.2}$$

Where

$$g = \text{length of generated solution}$$
$$r = \text{length of reference solution}$$

The final BLEU metric is then described as:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log\left(\text{p}_n\right)\right) \tag{4.3}$$

Where

$$\text{generally } w_n = \frac{1}{N}$$

ROUGE

ROUGE (Lin, 2004) is another metric based on n-grams and originally intended for the natural language domain, especially its variation ROUGE-L. It focuses on the longest common sub sequence between the proposed and reference solution, where longer sub sequences indicate closeness to the human standard.

ROUGE-L uses precision and recall

$$R(G,R) = \frac{\text{LCS}(G,R)}{\text{len}(R)}, P(G,R) = \frac{\text{LCS}(G,R)}{\text{len}(G)} \tag{4.4}$$

Where

$$\begin{aligned} G &= \text{Generated solution} \\ R &= \text{Reference solution} \\ LCS &= \text{Longest common sub sequence} \end{aligned}$$

to calculate the overall metric:

$$\text{ROUGE}_L(G,R) = \frac{2 \cdot P(G,H) \cdot R(G,H)}{P(G,H) + R(G,H)} \tag{4.5}$$

chrF

chrF (Popović, 2015) is a character-level metric intended for machine translation. It differs to the before seen metrics by using every character to create its measurement. It follows the same calculation as ROUGE-L but substitutes the n-gram context with character level.

$$\text{chrF}(G,R) = \frac{2 \cdot P_{\text{chr}}(G,H) \cdot R_{\text{chr}}(G,H)}{P_{\text{chr}}(G,H) + R_{\text{chr}}(G,H)} \tag{4.6}$$

Some papers use these method to claim the superiority of certain models over others. In the context of natural language, it is feasible to use such techniques to measure the effectiveness of language models. Likewise, such techniques are undoubtedly useful to measure language models built to generate code. Studies like (Evtikhiev et al., 2023) propose that evaluation metrics designed for natural language are applicable to code-generating models, but it is questionable to solely depend on such scores to benchmark the models and rank them against each other.

Drawing on the conclusions found in the section about the naturalness of code Section 2.1, there are theoretically significant arguments that metrics developed to evaluate natural language should only find limited use in the evaluation of generated code. For instance, comparing model capabilities on behalf of their BLEU score poses issues with semantic robustness. Changing or leaving out an important code element can lead to a higher BLEU score for one model over the other, while changing the functionality of the generated code. Such behavior favors a logically incorrect model over a correct one. Additionally, BLEU ignores the importance of individual words and therefore cannot account for keywords such as *function*, *class*, or *public*.

Logically, all evaluation metrics originating from the NLP domain are predominantly used for evaluating machine translation or other processes in natural language. All suffer from similar limitations when applied to models focused on code generation.

4.1.2 NLP Metrics & Naturalness of Code

Several metrics were introduced that address the shortcomings of BLEU and otherNatural Language Processing evaluation methods. These metrics use NLP as a stepping stone and account for some of the differences of text and code introduced in Section 2.1.

These metrics make use of specific representations of code such as:

Abstract Syntax Tree (AST)
AST is a hierarchical tree-like representation of the syntactic structure of code. It is commonly used in compilers and interpreters.

Abstract Semantic Graph (ASG)
ASG is a graph-based representation of the semantic meaning of code. It captures higher-level concepts and relationships beyond the syntactic structure represented by AST.
(Also called dependency graph or data flow in the subsequent sections)

codeBLEU

Ren et al. (2020) creates the methode codeBLEU and establishes a better correlation to the coding domain, by integrating Abstract Syntax Trees and Abstract Semantic Graphs, as well as an n-gram matches like BLEU does.

The AST representation of the generated and reference solution is used to calculate the syntactical closeness. It is an indicator for the code quality, since it catches syntactical errors such as missing tokens or identifier mismatches in the AST.

$$\text{Match}_{\text{ast}} = \frac{\text{count}(T_{\text{gen}})}{\text{count}(T_{\text{ref}})} \tag{4.7}$$

Where

$$\text{count}(T_{\text{gen}}) = \text{Number of sub-trees of the generated solution}$$
$$\text{count}(T_{\text{ref}}) = \text{Number of sub-trees of the reference solution}$$

The Abstract Semantic Graph (ASG) or data flow measures the semantic similarity and subsequently a way to measure semantic closeness between generated and reference solution.

$$\text{Match}_{\text{df}} = \frac{\text{count}(DF_{\text{gen}})}{\text{count}(DF_{\text{ref}})} \tag{4.8}$$

Where

$$\text{count}(DF_{\text{gen}}) = \text{Number of data flows of the generated solution}$$
$$\text{count}(DF_{\text{ref}}) = \text{Number of data flows of the reference solution}$$

The data flows are obtained by identifying all variables in the AST and representing them as nodes in the data flow graph. An edge is created between two nodes if one variable is created by or dependent on another. The resulting graph shows the relationships between all the variable of the code in question.

Finally the codeBLEU metric is mathematically described as:

$$\text{codeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}} \tag{4.9}$$

Where

$$\text{BLEU} = \text{Standard BLEU}$$
$$\text{BLEU}_{\text{weight}} = \text{BLEU with different weights} w_n$$
$$\text{Match}_{\text{ast}} = \text{Syntactic AST match}$$
$$\text{Match}_{\text{df}} = \text{semantic data flow or ASG match}$$

RUBY

RUBY (Tran et al., 2019) is an ensemble metric and focuses on tree similarity, graph similarity and string similarity. It is a three parted metrics that uses edit distance between generated and reference trees as the underlying method to calculate the differences.

String similarity:

$$STS(G,R) = \frac{SED(S_R, S_G)}{max(length(S_R), length(S_G))} \tag{4.10}$$

Where

$$S_R = \text{Reference string sequence}$$
$$S_G = \text{Generated string sequence}$$
$$SED(S_R, S_G) = \text{Edit distance between} S_R \text{ and } S_G$$

Tree similarity:

$$TRS(G,R) = \frac{TED(AST_R, AST_G)}{size(AST_R) + size(AST_G))} \tag{4.11}$$

Where

$$\begin{aligned} AST_R &= \text{Reference abstract syntax tree} \\ AST_G &= \text{Generated abstract syntax tree} \\ TED(S_R, S_G) &= \text{Edit distance between} AST_R \text{ and } AST_G \\ size(AST_X) &= \text{Number of nodes in a AST} \end{aligned}$$

Graph similarity:

$$GRS(G, R) = 1 - \frac{GED(PDG_R, PDG_G)}{size(PDG_R) + size(PDG_G))} \tag{4.12}$$

Where

$$\begin{aligned} PDG_R &= \text{Reference program dependence graph} \\ PDG_G &= \text{Reference program dependence graph} \\ ED(S_R, S_G) &= \text{Edit distance between} PDG_R \text{ and } PDG_G \\ size(PDG_X) &= \text{Number of vertexes in a PDG} \end{aligned}$$

Finally the ruby metric is the ensemble of the these three sub-metrics:

$$RUBY(G; R) = \begin{cases} GRS(G, R), \text{if } PDGs \\ RS(G, R), \text{if } ASTs \\ STS(G, R), \text{otherwise} \end{cases} \tag{4.13}$$

The presented metrics aim to tackle the challenge of code-specific evaluation by incorporating syntactic and semantic features of code, such as AST and ASG into their computation. These metrics represent a promising advancement in automated evaluation, leveraging characteristics from both natural language and code domains. However, a critical aspect remains unresolved: assessing whether the code fulfills its intended functionality.

4.1.3 Functional Correctness

Automated metrics like BLEU attempt to approximate the human standard and measure the similarity to the optimal code reference. However, the crucial property of functional correctness cannot be adequately addressed by either the original NLP metrics nor the more code-specific metrics discussed in Section 4.1.1 and Section 4.1.2.

Function-level

M. Chen et al. (2021) acknowledge the importance of evaluating and testing for functional correctness and introduce the evaluation metric HumanEval alongside the

Codex model, the precursor to Github Copilot. HumanEval is a manually-created metric, with originally 164 hand-crafted evaluation problems with an average of 7.7 unit tests for each. It is intended to evaluate the Codex, which is intended for the coding domain.

Similarly, Hendrycks et al. (2021) introduces APPS, another metric based on the same idea of measuring code by running unit tests against it. APPS represents a significantly wider accumulation of coding problems primarily based on open-source data from coding websites. The scope of the included problems has a wider range and integrates several difficulty levels.

Austin et al. (2021) produces another function-level metric called Mostly Basic Programming Problems (MBPP). MBPP consists of 974 entry-level Python functions with associated unit tests.

As seen in this section, multiple benchmarks follow the approach of implementing hand crafted testing frameworks. They deliberately are taking a step back from the automation principles that pushed the development of BLEU in the early 2000s. All in all, metrics such as HumanEval, MBPP or APPS are well established in the research communities. They are well positioned to fully evaluate function level code. But pushing the boundaries for code generating models beyond this initial stage will need larger scaled tasks such as classes or even repository/projects. Models should not only be trained on such use cases, but the evaluation metrics that are able to capture and assess them need to be created as well. Recent research has initiated the groundwork to start an evaluation in a larger context in the form of classes.

Class level

Du et al. (2023) recognizes the need to broaden functional correctness to larger scales and introduces the evaluation metric CLASSEVAL. This metric focuses on class-level evaluation and consists of 100 class-level problems for Python, each with the corresponding class-level testing. While the metric operates in the same manner as other metrics such as HumanEval, the scale differs considerably. HumanEval typically averages on 11.5 lines of code or 67.7 tokens, whereas CLASSEVAL averages on 45.7 lines of code and 259.3 tokens. Consequently, evaluating existing models using CLASSEVAL reveals notably lower performance in class generation tasks.

Table 4.1: *Existing evaluation methods and metrics*

Name	Measures	Metric	Ref
		NLP	
BLEU	Similarity	Precision	4.1.1
Rouge-L	Similarity	Precision, Recall	4.1.1
chrF	Similarity	Character level Precision, Recall	4.1.1
		NLP & Code Naturalness	
codeBLEU	Similarity	Precision	4.1.2
	Syntactic correctness	AST	
	Semantic correctness	Data Flow graph	
RUBY	String similarity	String Edit distance	4.1.2
	Syntactic correctness	AST Edit distance	
	Semantic correctness	Dependency graph Edit distance	
		Functional correctness	
HumanEval	Function-level correctness	Unit testing	4.1.3
APPS	Function-level correctness	Unit testing	4.1.3
MBPP	Function-level correctness	Unit testing	4.1.3
CLASSEVAL	Class-level correctness	Unit testing	4.1.3

4.1.4 Current state of evaluation metrics and the lack thereof

Numerous studies assess and compare different models based on their results of the above-mentioned metrics. Evtikhiev et al. (2023) does an exhaustive study on the validity on metrics originating from the NLP domain and finds them generally lacking for an accurate model comparison. Allamanis et al. (2018) already states in the beginning of the exploration of Transformer models in 2018 that the current suite of evaluation metrics for the code generation task is lacking.

They clearly encourages researchers to study new and appropriate manners to evaluate code. Functional correctness is the right direction, but it reaches its zenith in its current form. An evaluation metric remains valuable only as long as there is still potential for improvement in the models that are evaluated by it. Once a certain threshold is reached, the significance of evaluating a model based on that specific metric diminishes. Instead, novel methods for measuring improvement must be explored. For instance, lets assume a LLM achieving near-perfect functional correctness on the HumanEval metric. Further evaluation solely based on that functional correctness metric becomes less meaningful, prompting the need for alternative evaluation sets. Shifting to a different metric that adheres to similar principles or introducing new problems and testing cases only partially addresses the issue. There's a limit to the amount of such variation that can be introduced to make a significant difference for a model to still gain valuable insights from the metric.

As of the moment of the conception of this paper, models like GPT-4 or Googles Gemini can solve around 76% of HumanEval problems successfully (Gemini Team et al., 2023). Recently, Huang et al. (2023) demonstrated that a combination of GPT-4 models can successfully solve 96.3% of HumanEval's problems. While this achievement is impressive and marks a significant advancement, it also signals that evaluation metrics such as HumanEval have served their purpose well but are approaching the critical moment where they will not be able to push the advancement of the code generation task any further.

Current metrics still play a vital role in assessing and ranking existing models and will continue to do so for the future, especially as smaller and more efficient models find their role in the industry. Nonetheless there's a clear need for more appropriate metrics to accommodate front running models. Research efforts should be directed towards addressing larger problem domains, following examples such as CLASSEVAL, which identified potential areas for improvement in the functional correctness for class level. This widens the research field and encourages researchers to focus on more complex tasks with more dependencies. By pushing to surpass the current scope of single functions or statements in the evaluation research, the capabilities of code generating models will hopefully follow suite.

This section answered the question : *How do we measure the quality of generated code*. An overview of commonly used metrics can be found in Table 4.1.

4.2 Software Engineering metrics

This section explores the domain of traditional software metrics answering the second research question: *"Can traditional software engineering metrics be used to assess the quality of generated code?"*.

The research community recognizes the lack of well suited evaluation methods for code generating models and is turning towards alternatives, such as evaluation methods involving human-created techniques. HumanEval, APPS and MBPP are well known for prioritizing functional correctness on function level. This shift demonstrates the willingness of researchers to accept the additional efforts of manually create and administer evaluation frameworks, provided they enhance the quality of the evaluation.
Recently Huang et al. (2023) demonstrated unseen performance on HumanEval by solving over 96% of the provided problems and their unit tests. This progress shows that metrics concentrating on functional correctness of function level coding tasks such as HumanEval have potentially hit their ceiling regarding their usefulness to evaluate state of the art models and techniques. One way is to widen the scope of the generated code to class level, as proposed by Du et al., 2023 . While providing a way to further explore the evaluation based on functional correctness, class level coding tasks allow to integrate other quality measurements found in software engineering.

At this point concepts concerning the evaluation task by traditional software engineering are defined.

Quality indicator
A quality indicator is a higher-level concept that describes certain aspects of software. Generally such indicators are quantifiable through one or more underlying metrics.

Software metric
A software metric is the quantifiable value used to assess the quality of code. It normally is associated with one or multiple quality indicators.

This sections aims to showcase the feasibility of utilizing software metrics and their quality indicators as valid evaluation metrics for the code generation task.

4.2.1 Quality indicators

The study of software quality has produced multiple well-documented definitions of what constitutes good software. Concepts such as the SOLID principles (R. Martin, 2000) guide today's software engineering practices. This paper orients itself on these principles and leverages the underlying quality indicators of the aforementioned principles.

4.2.2 Extraction

In the following section the five SOLID principles (R. Martin, 2000) are further inspected and the correlating quality indicators are extracted and highlighted at the end of each paragraph.

Single Responsibility Principle (SRP)

"A class should have one, and only one, reason to change"

The principle ensures that a module or class only has one specific responsibility and that changes do not originate from the domain of other responsibilities.

Robert C. Martin clarifies and explains the principle in a blog post in 2014:

> *If you think about this you'll realize that this is just another way to define cohesion and coupling. We want to increase the cohesion between things that change for the same reasons, and we want to decrease the coupling between things that change for different reasons.* - R. C. Martin (2014)

Cohesion* | *Coupling

Open-Closed Principle (OCP)

"A module should be open for extension but closed for modification"

This principle states, that a module should be changeable without changing its current composition. The key for this principle is abstraction and can be done by using an interface for example.

Consider the following example:
A class animal has a function makeSound. For different animals the makeSound function should produce different sounds. Every time a new animal is created the animal class would need to be adjusted with the new sound for the new animal. Imagining 100 animals to be added. This would make this class undesirably hard to *maintain*. The Open-Closed Principle states that the original animal class should not be *modified*. Instead an animal should be able to implement its own makeSound functionality. This is achieved by making the animal class abstract, allowing others to *extend* its functionality. This drastically reduces *complexity* and an animal interface can be *reused* for an unrestricted number of different animals.

Modifiability* | *Extensibility* | *Maintainability* | *Complexity* | *Reusability

Liskov Substitution Principle (LSP)

"Subclasses should be substitutable for their base classes."

This principle states that that anyone who uses a class should also be able to be working with a subclass.

Example:
Consider an object of type cook that has the functionality prepareFood. PrepareFood uses an object of type animal. The Liskov Substitution Principle states that a cook should be able to continue to use prepareFood for any sub type of animal. As such the cook will be able to prepareFood with the derived type duck. This behavior makes the situation easier to *maintain*. The cook will be able to *extend* his capabilities to prepareFood by treating new kinds of animals without having to change his behavior specifically for a sub type.

Modifiability* | *Extensibility* | *Maintainability

Interface Segregation Principle (ISP)

"Many client specific interfaces are better than one general purpose interface "

The principle dictates to split up large interfaces. Objects relying on an interface should not be forced to rely on functionality they do not use.

Example:
The example restaurant can be separated into two services. The front facing one would be the customer service part. This is where a customer object orders, eats and pays for their meal. The other service is the back facing one where the cooks prepare the meals. With ISP the two services are clearly separated. Apart from getting the order, the cooking portion does not need any other interaction with customer. The two services can *change* things around without impacting the other one. For example would the rearranging of tables in the front facing service of the restaurant not impact the operation in the kitchen. *Maintenance* becomes easier since one can be addressed without influencing or touching the other.

Modifiability* | *Maintainability

Dependency Inversion Principle (DIP)

"Depend upon Abstractions. Do not depend upon concretions."

DIP promotes to inherit from abstract classes or interfaces and not from concrete one. The reason is simple, namely concrete class tend to change more frequently in comparison with abstract classes. Abstract classes are intended to provide common functionality and stability in software by design.

Example:
Assuming there are multiple different cooks, like a cook specialized on grilling on, one on steaming, etc. This would be implemented with an abstraction of base type cook and specializations would be deriving from it. A restaurant directly stating that is using a grilling cook would go against the Dependency Inversion Principle since the restaurant would be depending on a concrete type of cook. Consequently, *modifying* the restaurants cook type, for example when seasons change, would be challenging. In this scenario, each different restaurant with a different type of cook would need a separate definition. This makes the situation more *complex*. To easily *change* or *extend* the options in cooking a restaurant should depend on the abstract class cook. Relying of reusable restaurant setups and the usage of any sub type of cook.

Modifiability* | *Extensibility* | *Maintainability* | *Complexity

Other quality indicators

This study includes another important quality indicator. Readability is closely connected to code naturalness since it is and indicator for how fluently and understandable written code is. It follows certain conventions and is measured by a multitude of metrics. It will be referenced throughout the section about Traditional metrics 4.2.4 and Code smells 4.2.5.

Readability

It is worth mentioning that there are other quality indicators to consider, such as *scalability* or *efficiency*. While presenting interesting future research topics, the integration of such metrics is not in the scope of this paper.

4.2.3 Resulting indicators and considerations

Seven quality indicators have been extracted in the previous section. Collectively, these indicators contribute to standardizing and controlling the quality of software code in the OOP domain.

Table 4.2 lists the selection of indicators this paper focuses on. The set is intended to capture the most important, higher level programming concepts.

Table 4.2: *Quality indicators*

Indicator	Description
Complexity	Indicator on how complicated code is and how interconnected the components and classes are
Maintainability	Indicator for how easy it is to maintain given code. This is mostly an accumulation of one or more characteristics such as readability, modifiability, and extensibility
Readability	Indicator on how easy it is for a human to read code. It is a combination of structure, formatting and other general code naturalness indicators such as the choice of function names
Modifiability	Indicator on how easy changing code is
Extensibility	Indicator on how easy it extending code is
Cohesion	Indicator for the strength of the relationships among objects
Coupling	Indicator for the coupling among components and classess

4.2.4 Software metrics

Software metrics refer to well-researched code quality measurements with a history of wide practical application in the field of computer science. For example, McCabe (1976) introduced Cyclomatic Complexity (CC) nearly 40 years ago and it has been consistently utilized to assess software quality ever since. These metrics are described by a rigorous mathematical approach either with empirical measurements of code properties or the use of graph theory. These metrics are commonly employed in software engineering and offer a wealth of insights since they build the mathematical basis for the code quality indicators mentioned in the previous section.

This section presents candidates in the form of software metrics. All candidates must adhere to the following inclusion criteria:

All potential metric candidates must be:

1. deterministically quantifiable
2. assignable to at least one quality indicator Table 4.2

As during the extraction of quality indicators, the domain of OOP has been chosen to find viable metrics to directly assess the quality of code. The domain has been widely explored and operates clearly within the class level functionality. This presents a suitable problem domain for future coding models. This paper orientates itself on the references of the systematic mapping study conducted in Nuñez-Varela et al. (2017) . The study lists reference of traditional metrics in recent literature and highlights the relevancy of the chosen candidates.

The following paragraphs present the metrics with short explanations and mathematical formulas. A summary with mappings to quality indicators can be found in Table 4.3 . A large number of the following metrics originate from the following paper Chidamber et al. (1994) .

Cyclomatic Complexity (CC) McCabe (1976) firstly introduced this metric. It quantifies the number of possible paths the tree representation of a programs control flow also known as Abstract Syntax Tree. The calculation determines the difference between the number of edges and the number of nodes in the AST and adding the number of different program entry points.

$$CC = E - N + 2P \tag{4.14}$$

Where:

$$\begin{aligned} E &: \text{Number of edges} \\ N &: \text{Number of nodes} \\ P &: \text{Number of connected components (entry points)} \end{aligned}$$

It is an indicator for *complexity* and is one of the most used metrics in any given software measurement framework. It is applicable to higher level construct such as classes as well as lower level functions. It pinpoints sections in need of refactoring and influences directly other indicators such as *maintainability*, *readability* and *modifiability*. It further impacts development through its implications regarding testing complexity.

Weighted Methods per Class (WMC) This metric defines this complexity measurement metric leveraging other complexity metrics. It is an ensemble metric that assigns weights to each class function, for example, by calculating its CC (Equation 4.14). The subsequent sum of those weights indicates the overall complexity of a class.

$$WMC = \sum_{f \in C} CC(f) \tag{4.15}$$

Where:

$$\begin{aligned} f &: \text{Function} \\ C &: \text{Class} \\ CC(f) &: \text{Cyclomatic Complexity of } f \end{aligned}$$

Large and complex functions are hard to understand. The accumulation of such functions in a class is very time consuming in regards to *readability* and *maintainability*.

Coupling between Objects (CBO) Measures the dependencies between different classes based on attributes and methods. The higher the *coupling* rate of different classes the more interwoven the structure of the software. It indicates high *complexity* and dependency between classes and therefor heightened *maintainability*, *modifiability* and *extensibility* costs.

$$CBO(C) = \text{Number of other classes } C' \text{ the class } C \text{ is coupled with} \tag{4.16}$$

Lack of Cohesion in Methods (LOCM) Describes how strongly methods in a class or module relate to each other.

Consider a class with n methods

$$M_1, M_2, ..., M_n$$

For each of the methods we define the set of class properties used by that method as

$$\{I_i\} = \text{ set of class properties used by the method } i$$

Now we define the two sets P and Q that describe the set of methods that do not have any class property in common and respectively the set of methods that do.

$$P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$$
$$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$$

$$LOCM(C) = \begin{cases} |P| - |Q| & \text{, if } |P| > |Q| \\ 0 & \text{, otherwise} \end{cases} \tag{4.17}$$

In other words, metric calculates the difference between the number of method pairs that do not use the same class properties with the number of method pairs that do use the same properties.
Low LOCM describes the case where the methods do make use of the same properties and therefore follow the same purpose and work well together, i.e have a high *cohesion*. In the opposite case, high LOCM shows that a class is to *complex* with a high *maintainability* and should be considered for being split up.

Depth of Inheritance Tree (DIT) This metrics calculates the depth of the inheritance tree. The larger the tree in general the harder *extensibility* and *maintainability* for the classes within will become. Classes at the bottom tend to inherit a great number of functions and properties from their parent classes, making them impacting their *readability*.

$$DIT(C) = \text{Depth of class C in the inheritance tree} \tag{4.18}$$

Lines of Code (LOC) Counts the number of lines in different manners. This metric can be split up into Physical Lines of Code (PLOC) and Logical Lines of Code (LLOC). PLOC counts the total number of all lines, comments and blanks included. LLOC on the other hand counts only executable code.

$$PLOC = \text{total number of all lines} \tag{4.19}$$

$$LLOC = \text{counts only executable code} \tag{4.20}$$

$$LOC = PLOC + LLOC \tag{4.21}$$

These metrics are both insightful and simple. Generated code could be measured against a reference solution's number of lines of code to see how close it is to an optimal solution. More precisely, it can be used as an indication of the minimal amount of lines of code needed to solve a given problem. Too large numbers of lines of code can negatively impact *readability* of the code base.
PLOC in particular can have negative impacts regarding *complexity* and *maintainability*. Especially comments can make code more complicated and hard to read. For instance, when changes are made and comments are not or wrongfully updated.

Number of Children (NOC) Measures the amount of children a certain class has.

$$NOC(C) = \text{Number of sub classes } C' \text{ of a class } C \tag{4.22}$$

High numbers are indicators for high *coupling* and *complexity* as well as dependence of sub classes towards their parent class. This restricts *modifiability*. Additionally, it can be a indication of improper usage of inheritance and not clearly separated concerns

Response for a Class (RFC) Counts the number of other class functions called by a class function call.

$$RFC(C) = |\text{RS}| = |\{M\} \cup_{\text{all } i} \{R_i\}| \tag{4.23}$$

Where:

$$\begin{aligned} \text{RS} &: \text{Response set for the class } C \\ \{M\} &: \text{Set of all methods in the class } C \\ \{R_i\} &: \text{Set of methods called by another method } i \end{aligned}$$

In other words, the Response for a Class is the number of methods that can be invoked when another method or object is called. High numbers are an indication of high class *complexity* provoking heightened *maintainability*.

Number of Methods (NOM) counts the number of methods of a class.

$$NOM = \text{number of methods of class} \tag{4.24}$$

Indication for *complexity* and *maintainability*.

Number of Attributes (NOA) counts the number of attributes of a class.

$$NOA = \text{number of class properties/attributes} \tag{4.25}$$

Indication for *complexity* and *maintainability*.

Fan-Out (FO) Counts the number of outgoing connections of class.

$$FO = \text{number of outgoing connections} \tag{4.26}$$

High values indicate higher dependency towards other classes. This can be a marking of poor design and might elevate the *complexity* of the class. Furthermore adding *maintainability* overhead by introducing more complex test cases.

Fan-In (FI) Counts the number of incoming connections to a class.

$$FI = \text{number incoming connections} \tag{4.27}$$

High values indicate that this class is widely used and might present a central pillar of the code. These kinds of classes are to be treated with additional care, since changes in those classes tend to have ripple effects and lead to other changes. Having too many of such high FI classes influences *complexity* and *cohesion*. Additionally similar to NOC it has impacts on *modifiability*.

Table 4.3: *Software metric candidates*

Software metric	Quality indicator
Weighted Methods per Class (WMC)	Complexity, Maintainability, Readability
Coupling between Objects (CBO)	Coupling, Complexity, Maintainability, Modifiability, Extensibility
Lack of Cohesion in Methods (LOCM)	Complexity, Maintainability
Depth of Inheritance Tree (DIT)	Complexity, Extensibility, Maintainability, Readability
Lines of Code (LOC)	Readability
Physical Lines of Code (PLOC)	Readability
Physical Lines of Code (PLOC)	Complexity, Maintainability, Readability
Number of Children (NOC)	Complexity, Coupling, Modifiability
Response for a Class (RFC)	Complexity
Number of Methods (NOM)	Complexity, Maintainability, Readability
Cyclomatic Complexity (CC)	Complexity, Readability, Maintainability
Number of Attributes (NOA)	Complexity, Maintainability, Readability
Fan-Out (FO)	Cohesion, Complexity
Fan-In (FI)	Cohesion, Complexity, Modifiability

4.2.5 Code Smells

Code smells indicate potential design flaws in software. In that sense they are similar to traditional metrics that underlay software principles but are generally of smaller scope and have a smaller overall impact. Code smell measurements are mostly applied independently of the size of the code scale and context. As such it represents a versatile measuring utensil that can not only be applied to classes or modules but also to functions.

Refactoring: improving the design of existing code (1999) provides an extensive list of various code smells and how to refactor them. Several of these identified smells, as discussed in the subsequent paragraphs, are drawn from this source. Notably, code smells tend to have an impact on specific quality metrics, particularly *readability* and *maintainability* (Yamashita et al., 2012 ; Abbes et al., 2011).

This section presents candidates in the form of code smells. All candidates must adhere to the following inclusion criteria:

All potential code smell candidates must be:

1. deterministically quantifiable
2. assignable to at least one quality indicator Table 4.2

The calculation of some of these smells is straightforward, therefor mathematical explanations are only provided where necessary.

Long methods Indicates overly complex function. Does not represent a standalone metrics and rather uses Lines of Code to be calculated.

Parameter Count Counts the number of parameters of a function.

Nested conditions Counts the level nesting in a function or class.

Unused variables/methods Counts the number of dead code statements in the form of unused variables and functions.

Magic Numbers Highlights and counts hard coded numbers in functions.

Broken Windows Shows and counts the number of unused variables, commented-out code, or unreachable branches.

Code Duplication Measures and counts duplicated code lines.

Table 4.4: *Code smells*

Smell	Quality Indicator
Long methods	Readability, Maintainability
Parameter Count	Readability, Maintainability
Nested conditions	Readability
Unused variables/methods	Readability, Maintainability
Magic Numbers	Readability, Maintainability
Data clumps	Readability, Maintainability
Broken Windows	Readability
Code Duplication	Maintainability

4.2.6 Results and Considerations

This thesis proposes 14 software metrics and identifies various code smells that meet the criteria for evaluating the quality of generated code.

A potential evaluation method could look as follows: First, a collection of class-level coding tasks, along with their reference solutions, is created. Subsequently, a set of quality indicators from Table 4.2 is selected for this collection. For each task and quality indicator pair, the associated software metrics from Table 4.3 and Table 4.4 are computed and recorded alongside the reference solution.
During the evaluation process, a model under assessment has its generated code evaluated using the same quality indicators and software metrics. The closer the metric of the generated code aligns with the quality indicators of the reference solutions, the better the overall score of the model.

This approach might even hold additional benefits. While NLP evaluation or functional correctness assess only one aspect, traditional software engineering methods utilize quality indicators that enable simultaneous assessment of multiple code qualities. This broader scope offers a wider range of potential enhancement and comparison strategies between models, and can effectively highlight nuances and differences among models.

Consequently, the existence of these metrics support the positive answer to the research question: *Can traditional software engineering metrics be used to assess the quality of generated code*.

5 Discussion

5.0.1 Threads

Capabilities of models

The biggest threat to the validity of this paper is the fact that most open source models are not yet up to the task to compute solutions for high level specifications. The number of token that can be reliably created is restricted, especially in the open source and research communities that do not have the same resources as larger companies. Still, there is steady progress in the capabilities of code generating models with longer and wider context awareness. But it might still be too early for the application of an evaluation framework that is mostly based on the assumption of longer, and more complex code constructs, as seen in traditional software engineering and Object-Oriented Programming.

The role of naturalness

A second threat, which is not widely acknowledged in this paper, would be that the naturalness of code plays a greater role in the perceived quality of code than previously assumed. For instance, one facet of human-written code that highlights its organic nature and close resemblance to natural language is the adherence to coding conventions. These conventions typically follow either naturally emerging patterns or predefined rules, reflecting the inherent linguistic nature of coding.

Coding conventions depend on multiple factors and are hard to define and approach in a mathematical manner. They follow the idea of code naturalness in the sense that they manifest differently in different coding languages and can vary depending on who is using them. They can be compared to dialects in natural language. They differ from person to person but are mostly enforced in a grouping of developers that work together on the same code. Coding conventions play into the overall quality of software since the closer such syntactic guidelines are followed, the cleaner the code will be. This in turn promotes maintainability and readability. Research into the huge code base of Microsoft shows that approximately 18% of review threads/discussions in commits are mentioning some sort of coding conventions (Allamanis et al., 2014). Such coding conventions play a big role in the perception of high quality code in a real world scenario but cannot be adequately represented in a evaluation metric since it is not deterministically quantifiable.

5.0.2 Future research

Creation of the evaluation set

Experience and the general acceptance of HumanEval in the case of functional correctness show that manually setting up environments to use more specialized evaluation frameworks is feasible. While creating a large enough evaluation set is time consuming it is still perceived as worthwhile by the community. The same should consequently hold true for the integration of traditional software metrics measurements into the evaluation process. But the question on how such a framework should be constructed still remains. Is there potential in using existing tools and interfaces to analyze generated code or should new tools tailored for the model training process be crafted? Could existing refernce solutions from CLASSEVAL or HumanEval be used to calculate a basic reference solution for the traditional metrics approach? Or do we need humanly crafted evaluation sets specifically constructed for assessing quality?
It is the intent to delve deeper into further research and others are encouraged to do the same. This paper is meant provide a guideline and basis to create such frameworks that can effectively evaluate future models in the code generation task.

Ensemble framework

This paper acknowledges the validity and necessity of NLP and function correctness evaluation. It would be interesting to research an ensemble framework incorporating these evaluation techniques with the addition of software metrics.

The following proposed evaluation framework would describe a crude version of such an endeavor.

1. **Code Naturalness:**
   Evaluation of the code naturalness by utilizing commonly used NLP metrics.

2. **Functional Correctness:**
   Evaluation focused on functional correctness by utilizing a unit testing setup. Unit testing frameworks will need an environment to run and test the generated code. Preparing such evaluation environments is highly important since the safety of the generated code cannot be guaranteed. Maintaining strict isolation of the tested code is a necessary precaution, but can in turn produce considerable overhead in preparing the evaluation pipeline. While beeing more invovled such evaluation practices have found large acceptance in the research community

and is already considered to be an industrial standard and is used to evaluate newly created high performing models such as Gemini (Gemini Team et al., 2023).

3. **Traditional Software Metrics:**
   Future research and models are expected to address broader and more complex problem domains. Quality indicators such as maintainability, extensibility and readability are important aspects of code and need to be integrated in the evaluation process of code generating models. This can be achieved by utilizing the proposed metrics in this paper.

6 Conclusion

Large Language Models have faced a shortage of alternative evaluation methods for the code generation task, prompting recent research, including this paper, to introduce novel ideas and proposing potential frameworks. To advance code generation models effectively, it is essential to work with challenging evaluation metrics that drive research of the code generation task.
Present and past evaluation metrics for the code-generating task have predominantly prioritized NLP metrics and only recently transitioned into a more appropriate method of measuring functional correctness by unit testing.
However, as future models grow more capable in understanding and responding to larger problem domains and wider scopes, the current setup of evaluation metrics needs to be reconsidered. Pursuing the notion of semantic, syntactical, and functional correctness is essential, but it will not suffice to evaluate the code qualities at the class or project level. Programs of this scale demand different methods like the ones applied during the quality assessment of large-scale software. As such, the evaluation on the basis of quality indicators found in traditional software engineering should be considered.

This study finds that the role of software metrics has been largely ignored in the research of code generating models. As such, the incorporation of traditional software metrics in the evaluation process is proposed. Key findings suggest that traditional software metrics are well suited for this evaluation task. A number of widely accepted software metrics has been identified to describe the quality of generated code, providing suitable candidates for the integration into a evaluation framework aimed to asses current and future models.

This page intentionally left blank.

Bibliography

Abbes, Marwen et al. (2011). “An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 181–190. ISBN: 9780769543437. DOI: 10.1109/CSMR.2011.24.

Alec, Radford et al. (2019). “Language Models are Unsupervised Multitask Learners | Enhanced Reader”. In: *OpenAI Blog* 1.8, p. 9. URL: https://github.com/codelucas/newspaper.

Allamanis, Miltiadis et al. (Feb. 2014). “Learning NATURAL coding conventions”. In: 16-21-November-2014, pp. 281–293. DOI: 10.1145/2635868.2635883. arXiv: 1402.4182. URL: http://arxiv.org/abs/1402.4182%20http://dx.doi.org/10.1145/2635868.2635883.

Allamanis, Miltiadis et al. (2018). “A survey of machine learning for big code and naturalness”. In: *ACM Computing Surveys* 51.4. ISSN: 15577341. DOI: 10.1145/3212695. arXiv: 1709.06182. URL: http://learnbigcode.github.io/..

Austin, Jacob et al. (2021). “Program Synthesis with Large Language Models”. In: pp. 1–34. arXiv: 2108.07732. URL: http://arxiv.org/abs/2108.07732.

Barr, E. T. and P. Devanbu (2016). “The naturalness of software”. In: *Perspectives on Data Science for Software Engineering*. Elsevier, pp. 51–55. ISBN: 9780128042069. DOI: 10.1016/B978-0-12-804206-9.00010-6. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780128042069000106.

Brown, Tom B. et al. (2020). “Language models are few-shot learners”. In: *Advances in Neural Information Processing Systems* 2020-December. ISSN: 10495258. arXiv: 2005.14165.

Chen, Mark et al. (July 2021). “Evaluating Large Language Models Trained on Code”. In: arXiv: 2107.03374. URL: http://arxiv.org/abs/2107.03374.

Chen, Shouyuan et al. (2023). “Extending Context Window of Large Language Models via Positional Interpolation”. In: pp. 1–18. arXiv: 2306.15595. URL: http://arxiv.org/abs/2306.15595.

Chidamber, Shyam R. and Chris F. Kemerer (June 1994). “A Metrics Suite for Object Oriented Design”. In: *IEEE Transactions on Software Engineering* 20.6, pp. 476–493. ISSN: 00985589. DOI: 10.1109/32.295895. URL: http://ieeexplore.ieee.org/document/295895/.

Cho, Kyunghyun et al. (2014). *Learning phrase representations using RNN encoder-decoder for statistical machine translation*. DOI: 10.3115/v1/d14-1179. arXiv: 1406.1078.

Devlin, Jacob et al. (2019). “BERT: Pre-training of deep bidirectional transformers for language understanding”. In: *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the*

Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference. Vol. 1, pp. 4171–4186. ISBN: 9781950737130. arXiv: 1810.04805.

Driess, Danny et al. (2023). "PaLM-E: An Embodied Multimodal Language Model". In: *Proceedings of Machine Learning Research* 202, pp. 8469–8488. ISSN: 26403498. arXiv: 2303.03378.

Du, Xueying et al. (2023). "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation". In: *Proceedings of* 1. arXiv: 2308.01861. URL: http://arxiv.org/abs/2308.01861.

Evtikhiev, Mikhail et al. (2023). *Out of the BLEU: How should we assess quality of the Code Generation models?* Vol. 203. 1. Association for Computing Machinery. DOI: 10.1016/j.jss.2023.111741. arXiv: 2208.03133.

Gemini Team et al. (2023). "Gemini: A Family of Highly Capable Multimodal Models". In: arXiv: 2312.11805. URL: http://arxiv.org/abs/2312.11805.

Google AI Blog (n.d.). https://blog.google/technology/ai/long-context-window-ai-models/. Accessed: March 23, 2024.

Hendrycks, Dan et al. (2021). "Measuring Coding Challenge Competence With APPS". In: NeurIPS. arXiv: 2105.09938. URL: http://arxiv.org/abs/2105.09938.

Huang, Dong et al. (2023). "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation". In: arXiv: 2312.13010. URL: http://arxiv.org/abs/2312.13010.

Jiang, Albert Q. et al. (2024). "Mixtral of Experts". In: arXiv: 2401.04088. URL: http://arxiv.org/abs/2401.04088.

Jones, Karen Sparck (1994). "Natural Language Processing: A Historical Review". In: pp. 3–16. DOI: 10.1007/978-0-585-35958-8_1.

Knuth, Donald E. (1984). "Literate Programming." In: *Computer Journal* 27.2, pp. 97–111. ISSN: 00104620. DOI: 10.1093/comjnl/27.2.97.

Kolen, John F. and Stefan C. Kremer (2001). "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies". In: *A Field Guide to Dynamical Recurrent Networks*, pp. 237–243. DOI: 10.1109/9780470544037.ch14.

Lin, Chin-Yew (July 2004). "ROUGE: A Package for Automatic Evaluation of Summaries". In: pp. 74–81. URL: https://aclanthology.org/W04-1013.

Martin, Rc (2000). "Design principles and design patterns". In: *Object Mentor* c, pp. 1–34. URL: http://www.cogs.susx.ac.uk/users/ctf20/dphil%7B%5C_%7D2005/Photos/Principles%7B%5C_%7Dand%7B%5C_%7DPatterns.pdf%7B%5C%%7D5Cnhttp://scm0329.googlecode.com/svn-history/r78/trunk/book/Principles%7B%5C_%7Dand%7B%5C_%7DPatterns.pdf.

Martin, Robert C. (2014). *Single Responsibility Principle*. URL: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html (visited on 03/20/2024).

McCabe, T.J. (1976). "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4, pp. 308–320. DOI: 10.1109/TSE.1976.233837.

Nuñez-Varela, Alberto S. et al. (2017). "Source code metrics: A systematic mapping study". In: *Journal of Systems and Software* 128, pp. 164–197. ISSN: 01641212. DOI: 10.1016/j.jss.2017.03.044. URL: http://dx.doi.org/10.1016/j.jss.2017.03.044.

OpenAI Platform Documentation (n.d.). `https://platform.openai.com/docs/models/gpt-3-5-turbo`. Accessed: March 23, 2024.

Papineni, Kishore et al. (2002). "BLEU: A method for automatic evaluation of machine translation". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics* 2002-July.July, pp. 311–318. ISSN: 0736587X.

Popović, Maja (Sept. 2015). "chrF: character n-gram F-score for automatic MT evaluation". In: *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Ed. by Ondřej Bojar et al. Lisbon, Portugal: Association for Computational Linguistics, pp. 392–395. DOI: `10.18653/v1/W15-3049`. URL: `https://aclanthology.org/W15-3049`.

Refactoring: improving the design of existing code (1999). USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201485672.

Ren, Shuo et al. (2020). "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis". In: 1949.Weaver 1955. arXiv: `2009.10297`. URL: `http://arxiv.org/abs/2009.10297`.

Scholar MIAGE (n.d.). `https://scholar.miage.dev/`. Accessed: March 23, 2024.

Touvron, Hugo et al. (2023a). "Llama 2: Open Foundation and Fine-Tuned Chat Models". In: arXiv: `2307.09288`. URL: `http://arxiv.org/abs/2307.09288`.

Touvron, Hugo et al. (2023b). "LLaMA: Open and Efficient Foundation Language Models". In: arXiv: `2302.13971`. URL: `http://arxiv.org/abs/2302.13971`.

Tran, Ngoc et al. (2019). "Does BLEU score work for code migration?" In: *IEEE International Conference on Program Comprehension* 2019-May, pp. 165–176. DOI: `10.1109/ICPC.2019.00034`. arXiv: `1906.04903`.

Vaswani, Ashish et al. (2017). "Attention is all you need". In: 2017-December, pp. 5999–6009. ISSN: 10495258. arXiv: `1706.03762`.

Workshop, BigScience et al. (2022). *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. arXiv: `2211.05100`. URL: `http://arxiv.org/abs/2211.05100`.

Yamashita, Aiko and Leon Moonen (2012). "Do code smells reflect important maintainability aspects?" In: *IEEE International Conference on Software Maintenance, ICSM*, pp. 306–315. ISBN: 9781467323123. DOI: `10.1109/ICSM.2012.6405287`.