Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code?

BONAN KOU, Purdue University, USA
SHENGMAI CHEN*, Brown University, USA
ZHIJIE WANG, University of Alberta, Canada
LEI MA, The University of Tokyo, Japan and University of Alberta, Canada
TIANYI ZHANG, Purdue University, USA

Large Language Models (LLMs) have recently been widely used for code generation. Due to the complexity and opacity of LLMs, little is known about how these models generate code. We made the first attempt to bridge this knowledge gap by investigating whether LLMs attend to the same parts of a task description as human programmers during code generation. An analysis of six LLMs, including GPT-4, on two popular code generation benchmarks revealed a consistent misalignment between LLMs' and programmers' attention. We manually analyzed 211 incorrect code snippets and found five attention patterns that can be used to explain many code generation errors. Finally, a user study showed that model attention computed by a perturbation-based method is often favored by human programmers. Our findings highlight the need for human-aligned LLMs for better interpretability and programmer trust.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → **Natural language processing**;

Additional Key Words and Phrases: Code Generation, Large Language Models, Attention

ACM Reference Format:
Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code?. *Proc. ACM Softw. Eng.* 1, FSE, Article 100 (July 2024), 24 pages. https://doi.org/10.1145/3660807

1 INTRODUCTION

Large Language Models (LLMs) have made significant process on code generation in recent years [4, 5, 26, 27, 29, 32, 36, 43, 87]. A recent study [14] shows that GPT-4, the state-of-the-art LLM with 1.7 trillion parameters, can correctly solve 84% of the Python programming tasks from the HuamnEval [27] benchmark. Despite this great progress, it remains unclear why and how LLMs can generate correct code from natural language descriptions.

Model attention analysis is a common methodology to understand how a model works. It has been widely adopted in computer vision [31, 38, 42, 44, 59, 63, 80, 101] to investigate whether a model pays attention to the salient parts of an image when making a decision. In particular, recent studies find that aligning model attention with human attention can effectively enhance model

*This work was done when Shengmai Chen was an undergraduate student at Purdue University.

Authors' addresses: Bonan Kou, Purdue University, West Lafayette, USA, koub@purdue.edu; Shengmai Chen, Brown University, Providence, USA, shengmai_chen@brown.edu; Zhijie Wang, University of Alberta, Edmonton, Canada, zhijie.wang@ualberta.ca; Lei Ma, The University of Tokyo, Tokyo, Japan and University of Alberta, Edmonton, Canada, ma.lei@acm.org; Tianyi Zhang, Purdue University, West Lafayette, USA, tianyi@purdue.edu.

ACM 2994-970X/2024/7-ART100
https://doi.org/10.1145/3660807

performance [8, 34, 42]. For instance, Huang et al. [42] show that the performance of Conv-4-based image classification models can be increased by up to 23% when they are trained to align with human attention. Furthermore, previous studies also suggest that users have more confidence and trust in human-aligned models [13, 40, 75, 89]. For instance, Boggust et al. [13] find that users determine the trustworthiness of a model by checking whether the model makes predictions based on features they consider important.

These findings lead to an important scientific question for LLM-based code generation—*do LLMs attend to similar parts of a task description like human programmers in code generation?* We choose to compare model attention with human attention, since it can help us determine whether LLMs grasp the deep semantics in a task description like humans or whether they just learn superficial patterns from training data, which is known as a common problem in machine learning. Furthermore, by comparing human and model attention patterns, we seek to investigate whether the attention differences can be used to explain some code generation errors and inform new opportunities to improve LLMs for code generation.

To bridge the knowledge gap, we made the first effort to unveil the code generation process of LLMs by analyzing *which parts of human language an LLM attends to when generating code*. We present a large-scale study that examines the attention alignment between six LLMs and human programmers on two popular code generation benchmarks— OpenAI's HumanEval benchmark [27] and Google's MBPP benchmark [7]. The hypothesis is that LLMs should generate code based on salient words from an NL description similar to human programmers, rather than generating code based on trivial tokens such as prepositions and delimiters. Specifically, we investigated the following research questions in this study:

RQ1 To what extent is model attention aligned with human attention?
RQ2 Can attention explain errors of code generation models?
RQ3 What is the impact of different attention calculation methods on attention alignment?
RQ4 Which attention calculation method is most preferred by programmers?

Since none of the existing code generation benchmarks contain programmer attention information,we created the first programmer attention dataset for the programming tasks from HumanEval and MBPP (1,138 tasks in total). We captured programmers' attention by asking two experienced programmers to manually label words and phrases that they considered essential to solving each programming task. Their labels are validated by a third programmer. On the other hand, to capture model attention, we implemented and experimented with twelve different attention calculation methods in three categories—*self-attention-based*, *gradient-based*, and *perturbation-based*. To ensure our findings generalize across different LLMs, we analyzed the attention of six LLMs with different sizes, including GPT-4 [2], InCoder-1.3B [32], CoderGen-2.7B [62], PolyCoder-2.7B [91], CodeParrot-1.5B [1], and GPT-J-6B [87].

Our study reveals several important insights into the code generation process of LLMs. First, we find a consistent *attention misalignment* in all six LLMs, regardless of the attention calculation methods. Furthermore, we performed an in-depth analysis of the attention patterns of 211 incorrect codes generated by the two best models in our study, CodeGen-2.7B, and GPT-4. We found that 27% of the code generation errors could be explained by five attention patterns. Finally, perturbation-based methods generated attention scores that are overall more aligned with human attention than other methods. They are also preferred by human programmers, according to a user study of 22 participants. Our findings highlight the need to develop human-aligned LLMs and provide practical guidelines for improving LLM-based code generation and calculating model attention.

In summary, this paper makes the following contributions:

- We conducted the first empirical study on the attention alignment of LLMs and human programmers on code generation tasks.
- We conducted a comparative analysis of different attention calculation methods for code generation models through both quantitative experiments and a user study.
- We made publicly available the first programmer attention dataset of 1,138 Python tasks, which can be used to develop new human-aligned models and evaluate interpretability methods for code generation models. Our code and data have been made available in our GitHub repository [47].

2 MOTIVATION AND PRELIMINARIES

In this section, we first explain the motivation for performing attention analysis for code generation models. Then, we introduce popular benchmarks and metrics for code generation. Finally, we define model attention and describe different kinds of attention calculation methods for LLMs.

2.1 Motivation

Model attention analysis is a common task in several domains, such as computer vision [31, 44, 59, 63], neural machine translation [18], and autonomous driving [38, 46, 80]. Specifically, several studies show that aligning model attention with human attention during the training process can significantly improve the model performance [8, 34, 42]. For instance, Huang et al. [42] show that by aligning the attention of Conv-4-based and ResNet-based models to human attention on images, the performance of these models on image classification can be increased by up to 23% in the one-shot setting and 10% in the five-shot setting. In autonomous driving, many studies have demonstrated that adding attention alignment constraints can help the autonomous driving system to drive safer [38, 46, 80]. This motivates us to investigate the attention alignment between LLMs and human programmers in the domain of code generation.

Several recent studies analyzed the attention of neural models for code summarization, program repair, and method name prediction [8, 64, 68]. Paltenghi et al. [64] studied the attention alignment between neural model attention and programmer attention on code summarization. Bansal et al. [8] showed that aligning the attention of neural code summarization models with human attention can effectively improve model performance. Rabin et al. [68] found that pre-trained code models rely heavily on just a few syntactic features in the prompts to perform method name prediction and variable misuse detection. To the best of our knowledge, none of the existing studies have investigated code generation tasks or LLMs with billions of parameters. Our study fills this gap by analyzing the attention patterns of six LLMs in code generation tasks.

Finally, another motivation behind this study is the fact that there is still no consensus on how to calculate the attention score of LLM-based code models. This is largely attributed to the complexity of the multi-head, multi-layer attention mechanism adopted by these models. For example, some studies only considered model attention from the first transformer layer and stated that the first layer encodes lexical-level information [10, 97], while other studies summed up the attention from all transformer layers to incorporate long-distance token relationships [56, 85]. To bridge this gap, we conducted the first comprehensive study on 12 attention calculation methods and systemically evaluated them with quantitative experiments and a user study with 22 participants.

2.2 Code Generation Benchmarks and Metrics

In this work, we focus on code generation tasks that generate a function from a natural language description. Since the breakthrough of OpenAI's Codex model in 2021 [27], this kind of code generation task has become increasingly popular in the research community. In this task setting, given a function header and a task description in natural language, an LLM is expected to complete the function according to the task description. Figure 1 shows an example.

codegen_2b_mono_output.py

```
from typing import List

def has_close_elements(numbers, threshold) -> bool:
    """
    Check if there are two numbers in a list that are
    closer to each other than a given threshold.
    """
+   for i in range(len(numbers) - 1):
+       if numbers[i + 1] - numbers[i] < threshold:
+           return True
+   return False
```

Fig. 1. A Python function generated by CodeGen-2.7B [62]. The generated code is highlighted in green.

Code generation models are often evaluated on crowd-sourced programming benchmarks. OpenAI developed a programming benchmark called HumanEval and used it to evaluate the original Codex model and its variants [27]. HumanEval [27] includes 164 Python programming tasks and ground-truth solutions. It is by far the most popular benchmark and has been used to evaluate most LLM-based code generation models. Besides, MBPP is a large benchmark [7] with 974 crowd-sourced programming tasks and solutions. Both HumanEval and MBPP include test cases to evaluate the functional correctness of generated code.

Two types of metrics are often used to evaluate the performance of LLM-based code generation models. First, if a code generation benchmark includes test cases, one can simply run the test cases to evaluate the correctness of the generated code. A typical metric in this category is *Pass@k*. *Pass@k* was initially introduced by Kulal et al. [50]. It measures the percentage of correctly solved programming tasks where k refers to the number of code samples generated by the LLM. If any of the k samples pass all test cases in a task, the task is considered correctly solved. OpenAI then introduces an unbiased version of *Pass@k* to reduce variances, which is widely used to evaluate code generation models these days [27]. In practice, many programming tasks do not have existing test cases and some code solutions do not have a well-defined function interface for testing, e.g., a single line of code without clear input and output. Thus, prior work also measures the similarity between generated code and a ground-truth solution as a proxy for correctness. BLEU [65] and CodeBLEU [69] are commonly adopted similarity metrics. BLEU is a metric commonly used to evaluate the quality of machine-generated text. It evaluates the quality of machine-generated text by comparing the presence of n-grams in the generated text to those in reference texts. BLEU calculates a score between 0 and 1, with 1 indicating perfect similarity. CodeBLEU is designed to adapt BLEU specifically for evaluating generated code. While BLEU primarily considers word-level similarity, CodeBLEU considers the structure and correctness of the generated code, which are crucial aspects in code generation tasks.

2.3 Model Attention

In this work, we use *model attention* to refer to how important a token in a natural language task description is considered by an LLM during code generation. It implies which parts of the input the model "attends" to during code generation. This idea resembles *feature importance* [41], *saliency map* [61], and *feature attribution* [60] in the XAI literature. We describe three kinds of model attention calculation methods as follows.

2.3.1 Self-Attention-Based Methods. The self-attention mechanism allows transformers to weigh the importance of different parts of the input when making predictions. By focusing on the most

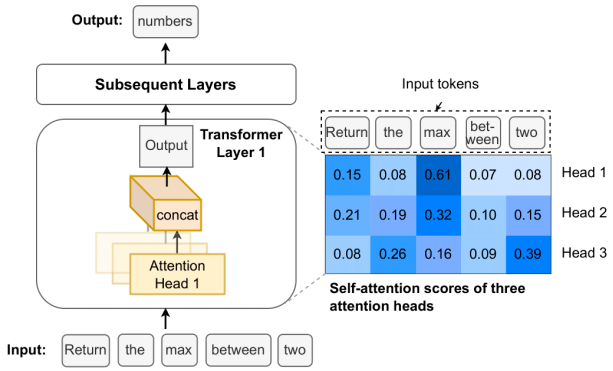

Fig. 2. Attention matrix of the first attention head of the transformer layer in CodeGen-2.7B

relevant tokens with the highest self-attention scores, the transformers can make better predictions by capturing relationships and dependencies in the input.

An LLM includes multiple *transformer layers*, each of which includes multiple *attention heads*. These attention heads independently calculate self-attention scores between different input tokens. Therefore, a transformer model could have multiple sources of model attention from different transformer layers and different attention heads in each layer. For example, Figure 2 shows the self-attention scores of the first three attention heads of the first transformer layer in CodeGen-2.7B when generating token *"numbers"* from input sequence *"Return the max between two"*. The self-attention scores calculated by different attention heads differ for the same token. Different heads represent different kinds of "focus" from the model.

To calculate model attention on the input sequence, vectors of self-attention scores from different attention layers and different attention heads in each layer need to be aggregated into a single vector to represent the overall importance of each input token to the model prediction. However, although self-attention scores have been generally used as model attention [20, 33, 52, 84], there is still no consensus on how to aggregate self-attention scores from different layers and attention heads. For example, some studies sum up the attention scores attention from all transformer layers and all attention heads in each layer [99] as the final attention scores. These studies argue that this summation strategy can capture long-distance token relationships. Some other studies only use the attention scores from the first transformer layer and argue that the first layer captures lexical-level dependencies [10, 97].

To reveal how different ways to aggregate self-attention affect the alignment between model and human attention, we experimented with six self-attention-based methods in this study (detailed in Section 4.2.1).

2.3.2 Gradient-Based Methods. Gradient-based methods leverage the gradients of the model's predictions concerning the input features to calculate the model's attention. The calculation of gradient-based methods includes two different steps: (1) perform a forward pass of input of interest, and (2) calculate gradients using backpropagation through the neural network's layers. By analyzing the magnitudes of these gradients, these methods can identify which input tokens are most influential in determining the output. For example, *Integrated Gradients* is a gradient-based method that computes the integral of the gradients of the model's output concerning each input feature [23, 76, 81]. In this study, we experimented with two gradient-based methods used in previous work [76, 78] with details in Section 4.2.2.

2.3.3 Perturbation-Based Methods. Different from the previous two categories of methods, perturbation-based methods [84, 90] are model-agnostic. In other words, they do not require access to the internal information of a model. Perturbation-based methods are particularly useful for calculating the attention of commercial models such as GPT-4, since these models do not reveal their self-attention layers or gradients to users.

Perturbation-based methods first mutate the input and then calculate the model's attention based on the output differences. LIME [70] and SHAP [57] are two popular perturbation-based methods. LIME [70] generates a local explanation by approximating the specific model predictions with a simpler model (e.g., a linear classifier). SHAP [57] enhances LIME by perturbing the input based on game theory and uses Shapely value to estimate different tokens' importance.

A limitation of these two methods is that they often require a large number of perturbed samples to ensure estimation accuracy. This is costly for calculating the attention for GPT-4, since we need to query GPT-4 many times. Furthermore, LIME and SHAP only mutate an input by deleting tokens, which may significantly change the meaning or the structure of an input. To address this limitation, more recent perturbation-based methods choose to substitute tokens with similar or semantically related tokens in the context [54, 90]. They often use a masked language model such as BERT [25] to predict similar or semantically related tokens to substitute existing tokens in an input. Then, they measure the influence of these substitutions on the output. In this study, we experimented with SHAP [57] and the BERT masking-based method [90] (detailed in Section 4.2.3).

3 THE CONSTRUCTION OF THE PROGRAMMER ATTENTION DATASET

Since none of the existing code generation benchmarks contain programmer attention information (i.e., which words or phrases a programmer considers important when writing code), we created the first programmer attention dataset based on the 1,138 programming tasks, including all 164 prompts from HumanEval [27] and the 974 prompts from MBPP [7]. We selected these two datasets, since they are widely used to evaluate code generation models and they also provide test cases for each programming task, which is important for calculating the correctness of model-generated code.

The first two authors, who have more than five years of programming experience in Python, manually labeled the words and phrases they considered important to solve the programming task in each task description. Before the labeling process, the two labelers went through programming tasks in HumanEval to familiarize themselves with the programming tasks and the code solutions. Then, they first independently labeled the first 20 task descriptions in HumanEval. This first round of labeling had a Cohen's Kappa score of 0.68. The two labelers discussed the disagreements and summarized four kinds of keywords that both of them considered important. The four keyword types are summarized below:

- **Data types**: Words or phrases that describe the types of data that the code should input or output, such as *"string"*, *"number"*, or *"list"*.
- **Operators**: Words or phrases that describe the operations that the code should perform on the data, such as *"compare"*, *"sort"*, *"filter"*, or *"search"*.
- **Conditionals**: Words or phrases that specify the conditions under which the code should execute, such as phrases after *"if"* and *"when"* in a task description.
- **Properties**: Important properties of the manipulated data and operations, such as quantifiers (e.g., *"all"*, *"one"*), adjectives (e.g., *"first"*, *"closer"*), and adverbs (e.g., *"every"*, *"none"*).

Although there are only four types of keywords, each type is designed to be high-level and inclusive. For instance, the "operator" type refers to any kind of operation on the data, such as *"sort a list"*, *"connect a database"*, and *"plot a graph"*. With this labeling standard, the two labelers

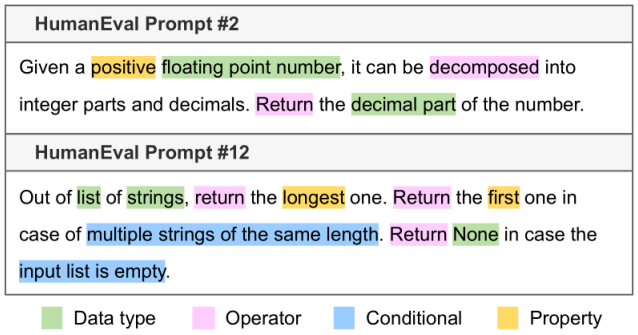

Fig. 3. Two examples of labeled prompts from our dataset.

proceed to label the remaining 144 task descriptions in the HumanEval dataset. The Cohen's Kappa score of this round of labeling increased to 0.72, indicating a substantial agreement [58]. Then, they discussed and resolved all disagreements.

To verify these labels, the third author, who was not involved in the previous labeling process, independently labeled the 164 task descriptions from the HumanEval dataset. Since Cohen's Kappa can only calculate the agreement level between two labelers, we used Fleiss' Kappa to measure the agreement between the third labeler and the initial labels from the first two labelers. The Fleiss' Kappa score is 0.64, indicating a substantial agreement [30]. This result shows labels made by the first two labelers are reasonable and can be accepted by other programmers. Then, the first two labelers continued to label 974 programming tasks in the MBPP dataset independently, which resulted in a Cohen's Kappa score of 0.73. Finally, they resolved all disagreements and used the final set of labels as the programmer's attention dataset. The entire labeling process takes 192 person-hours.

On average, each task description has 29.6 words, among which 7 words are considered important by both labelers. Among all four types of keywords, *property* keywords (45.2%) are labeled the most frequently by the two labelers, followed by *operators* (27.2%), *conditional* (25%), and *data types* (2.6%). Figure 3 shows two examples of labeled task descriptions. The four types of keywords are labeled in different colors.

4 METHODOLOGY

This section describes the study design to answer the research questions listed in Section 1.

4.1 Code Generation Models

In this study, we select six LLMs with different sizes and different model performances on code generation tasks. Table 1 shows the size, the number of self-attention layers, the number of attention heads in each layer, and the model performance on the combined dataset of HumanEval and MBPP in terms of Pass@1. We describe each model below.

- **InCoder-1.3B [32]** is an open-source code language model from Meta AI. It is trained on 159 GB of permissively licensed code from GitHub, GitLab, and Stack Overflow. Compared with other LLMs, it adopts a new causal masking objective, which allows it to infill blocks of code conditioned on the arbitrary left and right contexts. We used the largest pre-trained model released by Meta, including 1.3B parameters and 24 transformer layers.
- **PolyCoder-2.7B [91]** is an open-source model from CMU. It is based on the GPT-2 architecture and is designed to be the open-source counterpart of OpenAI Codex [27], since Codex is not open-sourced. It is trained on 249GB of code and has 2.7B parameters.

Table 1. Code generation models included in this study.

Model	Layer	Head	Pass@1
InCoder-1.3B	24	32	15.20%
PolyCoder-2.7B	32	32	5.59%
CodeGen-2.7B	32	32	23.70%
CodeParrot-1.5B	48	25	3.58%
GPT-J-6B	28	16	11.62%
GPT-4	-	-	67%

- **CodeGen-Mono-2.7B [62]** is an open-source model from Salesforce Research. It follows a standard transformer autoregressive model architecture with rotary position embedding.
- **CodeParrot-1.5B [1]** is another open-source effort of training a GPT-2 model for code generation. It is trained on 180GB Python Code and has 1.5B parameters.
- **GPT-J-6B [87]** is an open-source model from EleutherAI. It adopts a transformer architecture similar to GPT-3. It is trained on 825 GB text data, which includes 95GB code from GitHub.
- **GPT-4 [2]** is the state-of-the-art language model developed by OpenAI. Since the internal structure of GPT-4 is not publicly disclosed, we do not include the number of layers and heads of GPT-4 in Table 1. It is reported that GPT-4 has about 1.76 trillion parameters [3]. We used the API (`gpt-4`) provided by OpenAI to query GPT-4.

4.2 Model Attention Calculation

We experimented with twelve attention calculation methods from three different categories: six *self-attention-based* methods, four *gradient-based* methods, and two *perturbation-based* methods.

4.2.1 Self-Attention-Based Methods. Given that LLMs have multiple attention layers, there is currently no consensus on what is the right way to aggregate those self-attentions to explain LLMs. Zeng et al. [97] show that the first attention layer is indicative of which tokens the model attends to, while Wan et al. [85] show that deeper attention layers are better at capturing long-distance dependencies and program structure. To perform a comprehensive analysis, we decide to experiment with three settings: (1) only using the first attention layer (denoted as *first*), (2) only using the last attention layer (denoted as *last*), and (3) using all attention layers (denoted as *all*).

To aggregate self-attentions across different attention heads in a layer, we follow the previous work [99] by summing the attention values from different heads. Finally, since LLMs generate code in an autoregressive manner, their attention changes in each step as they read more tokens from the input and as they generate more code. We are curious about which input tokens the model highly attends to as they read the input and which input tokens the model highly attends to as they generate code. So we consider two experiment settings: (1) summing up the attention scores assigned to each token during the input reading process (denoted as *READING*), and (2) summing up the attention scores assigned to each token during the code generation process (denoted as *CODING*). Given the three settings in layer-wise attention aggregation and the two settings in step-wise attention aggregation, we have a total of six experiment settings: *READING_first*, *CODING_first*, *READING_last*, *CODING_last*, *READING_all*, and *CODING_all*.

4.2.2 Gradient-Based Methods. We consider two different methods to calculate gradient-based model attention: (1) *Saliency* [78], and (2) *Input×Gradient* [76]. The saliency method calculates the model's attention by computing model gradients with respect to the input. Given a LLM $\mathcal{F}$, suppose an input is $X = [x_1, x_2, \ldots, x_n]$, where n is the length of the input. The attention s_i on x_i is

calculated as $s_i = \frac{\partial \mathcal{F}(X)}{\partial x_i}$. Different from the saliency method, *Input×Gradient* further multiplies gradients with the input's embedding values. The attention s_i on x_i is calculated as $s_i = x_i \cdot \frac{\partial \mathcal{F}(X)}{\partial x_i}$.

Similar to self-attentions, gradients also change constantly at each generative step. Thus, we also experimented with the two step-wise attention aggregation settings as in the self-attention methods. The combination of the two gradient calculation methods with the two step-wise aggregation settings results in four gradient-based methods—*Input×Gradient_reading*, *Input×Gradient_coding*, *Saliency_reading*, and *Saliency_coding*.

4.2.3 Perturbation-Based Methods. We consider two different perturbation-based methods: (1) *SHAP* [57], which masks the input through deleting some tokens, and (2) *BERT Masking* [90], which masks the input through substituting a token with its masked language modeling prediction result from BERT.

- *SHAP*. We use the official SHAP library with a perturbation count equal to 50 to calculate the model's attention (SHAP scores) on different tokens.[1] We aggregate SHAP scores on predicting different tokens by summing them up.
- *BERT Masking*. Since the code from the original BERT Masking paper [90] is not publicly available, we re-implemented this method based on the description in the paper. Specifically, given each token in an input prompt, this method masks it and uses a pre-trained BERT model from HuggingFace to predict the most likely token in the masked position. Then, it substitutes the original token with the predicted token and prompts the LLM to regenerate the code solution. This method then calculates the model's attention on this token by calculating the BLEU score [65] between the original code solution and the new solution. We iterate through all tokens in the input prompts to obtain model's attention on different tokens.

4.3 Attention Alignment Measurement

We measured the attention alignment between models and human programmers using two robust metrics—San Martino's token overlapping metrics [22] and Krippendorff's alpha [49]. We also experimented with two simple metrics, Cohen's kappa and keyword coverage rate, and obtained similar results. Due to the page limit, we only reported the results of the first two metrics in this paper. The results of the other two metrics are included in the GitHub repository [47].

4.3.1 San Martino's Token Overlapping Metrics [22] calculate the overlap between two sets of tokens in terms of precision, recall, and F-1 score. It was initially designed for sequence labeling tasks in NLP [22] and has been recently adopted in SE studies as a robust metric for toxicity detection in code review comments [72]. Specifically, it assigns partial credit for partial overlaps between two sets of tokens, which makes it a suitable choice for our task. Thus, we follow [22] to compare salient words selected by humans and models. For human attention, a token is considered *highly attended* if it is labeled as an important word by human programmers, as described in Section 3. For model attention, since the attention calculation methods in Section 4.2 compute a continuous score for each token, it is hard to determine a universal threshold to decide which token is highly attended by a model. Thus, we rank the tokens in a programming task description based on their attention scores and select the top K tokens as the *highly attended* tokens by a model. Since the average of human-labeled important words is 7 per task description, we experiment with $K = 5, 10, 20$ in our study. Given a set of highly attended tokens by human programmers and a set of highly attended tokens by a model, we follow the equations in [72] to compute precision, recall, and F-1 score. We report all three scores in Table 2.

[1] https://shap.readthedocs.io/en/latest/

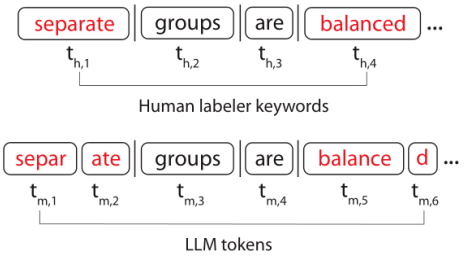

Fig. 4. Mapping NL words to LLM sub-tokens

4.3.2 Krippendorff's Alpha [48] is a robust statistical measure for inter-labeler agreement, where $\alpha = 1$ indicates perfect agreement, $\alpha = 0$ indicates no agreement, and $\alpha = -1$ indicates complete disagreement. It can be used to measure the agreement level between human programmers and LLMs on important words in a programming task description. Compared to other inter-rater agreement metrics such as Cohen's kappa [21], Krippendorff's alpha is more robust to the number of coders, missing data, and sample size. To calculate Krippendorff's alpha, human and model labels must be stored in vectors of the same length. This is hard because LLMs perform subword tokenization to handle out-of-vocabulary words with a manageable vocabulary size. For example, in Figure 4, the word "separate" is tokenized into two tokens: "separ" and "ate" through byte pair encoding (BPE). During inference, an LLM will compute attention scores separately for these two subtokens, though they are from the same English word.

To address this challenge, we map NL words back to LLM tokens. If the model tokenizes a natural language word into multiple tokens, all sub-tokens will be considered selected by human labelers. Using the same example from Figure 4, if the natural language word "separate" is selected by the human labelers, both sub-tokens "separ" and "ate" will be considered selected and represented by 1 in the vectors used to calculate Krippendorff's alpha.

4.4 User Study Design

To answer RQ4, we conducted a user study to evaluate the different attention calculation methods.[2] We recruited 22 students (18 males and 4 females) through the department mailing lists in a CS department. According to our user study, all participants have an average of 5.62 years of programming experience. All participants have some basic understanding of model attention in machine learning.

We randomly selected 8 task descriptions from our dataset. For each task, we leveraged CodeGen-2.7B, the best-performing open-source model on HumanEval in our experiment, to calculate its model attention. We did not consider GPT-4 in this user study, since it is close-sourced and we cannot compute its attention using self-attention-based methods and gradient-based methods. We selected one attention calculation method from each category (*self-attention-based*, *perturbation-based*, and *gradient-based*): $CODING\_last$, $Input \times Gradient\_coding$, and *SHAP*.

In each user study, participants first read the task description to understand the programming task and then read the code generated by CodeGen. For each attention method, we render a highlighted version of the task description similar to Figure 3, where the top 10 attended tokens are highlighted based on the attention scores computed by this method. We chose to render the top 10 important keywords since it is close to the average number of important words (7) labeled in our dataset.

[2]Our user study questionnaire and participants' responses are available in our GitHub repository: https://github.com/BonanKou/Attention-Alignment-Empirical-Study.

Table 2. Human-model attention alignment calculated by different methods in terms of San Martino's Precision (P), Recall (R), F1 score (F1), and Krippendorff's alpha score (KA). Among the 12 different attention calculation methods, the one that produces the most aligned result for each model under each K setting in each metric is highlighted in yellow. Note that we only experimented with perturbation-based methods on GPT-4, since GPT-4 does not provide access to its self-attention layers and gradients.

(a) Perturbation-based methods.

Method	BERT_masking												SHAP											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	48.8%	40.6%	44.3%	0.25	40.8%	67.4%	50.8%	0.25	36.6%	90.3%	52.1%	0.12	33.9%	28.4%	30.9%	0.11	33.4%	55.9%	41.8%	0.18	32.8%	86.7%	47.6%	0.15
CodeGen	51.8%	43.2%	47.1%	0.29	41.8%	68.9%	52.0%	0.27	36.8%	90.7%	52.4%	0.13	33.0%	27.5%	30.0%	0.10	32.9%	54.9%	41.1%	0.17	33.0%	87.1%	47.9%	0.16
CodeParrot	51.5%	43.0%	46.9%	0.29	42.1%	69.2%	52.3%	0.28	36.6%	90.2%	52.0%	0.12	33.2%	27.7%	30.2%	0.10	32.7%	54.7%	40.9%	0.17	33.3%	87.6%	48.3%	0.17
GPT-J-6B	49.9%	41.3%	45.2%	0.26	41.0%	67.4%	51.0%	0.26	36.6%	90.3%	52.1%	0.12	33.3%	27.8%	30.3%	0.10	33.3%	55.7%	41.7%	0.18	33.1%	87.1%	47.9%	0.16
PolyCoder	49.3%	41.3%	45.0%	0.26	40.9%	67.7%	51.0%	0.26	36.6%	90.4%	52.1%	0.12	34.4%	28.9%	31.4%	0.12	33.5%	56.2%	42.0%	0.19	33.2%	87.2%	48.1%	0.16
GPT-4	32.7%	27.6%	30.0%	0.04	36.7%	61.7%	46.0%	0.17	36.3%	89.4%	51.7%	0.11	34.7%	29.2%	31.7%	0.12	34.2%	57.4%	42.9%	0.20	33.1%	87.0%	47.9%	0.16

(b) Gradient-based methods

Method	Input×Gradient_reading												Input×Gradient_coding											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	38.4%	30.3%	33.9%	0.07	42.8%	68.7%	52.8%	0.30	41.6%	89.1%	56.7%	0.31	41.0%	32.5%	36.3%	0.10	43.9%	69.0%	53.7%	0.31	41.6%	89.1%	56.8%	0.31
CodeGen	44.8%	34.2%	38.8%	0.22	39.4%	59.8%	47.5%	0.28	35.2%	87.3%	50.1%	0.22	46.0%	34.7%	39.5%	0.23	41.0%	61.5%	49.2%	0.31	36.2%	88.5%	51.4%	0.25
CodeParrot	55.0%	43.2%	48.4%	0.33	45.8%	71.2%	55.8%	0.40	35.1%	87.5%	50.1%	0.22	57.9%	44.9%	50.6%	0.37	46.8%	71.8%	56.7%	0.42	36.0%	88.7%	51.2%	0.24
GPT-J-6B	40.9%	30.7%	35.1%	0.17	37.4%	56.7%	45.1%	0.24	34.5%	85.8%	49.2%	0.20	42.7%	32.5%	36.9%	0.19	39.4%	59.6%	47.4%	0.28	35.5%	87.2%	50.5%	0.22
PolyCoder	43.3%	33.8%	38.0%	0.20	38.3%	59.9%	46.7%	0.26	34.4%	86.2%	49.1%	0.20	38.1%	30.1%	33.6%	0.14	36.4%	57.3%	44.5%	0.23	35.2%	87.4%	50.2%	0.22
Method	Saliency_reading												Saliency_coding											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	36.6%	29.3%	32.6%	0.06	41.9%	67.3%	51.6%	0.29	41.4%	88.6%	56.4%	0.30	39.4%	31.5%	35.0%	0.09	42.8%	67.3%	52.3%	0.29	41.3%	88.5%	56.4%	0.30
CodeGen	46.1%	34.9%	39.8%	0.23	39.3%	59.6%	47.4%	0.28	35.3%	87.4%	50.3%	0.23	47.3%	35.4%	40.5%	0.24	40.7%	60.8%	48.8%	0.30	36.2%	88.5%	51.4%	0.25
CodeParrot	53.8%	42.1%	47.2%	0.31	44.6%	69.3%	54.3%	0.38	34.8%	86.8%	49.7%	0.21	56.3%	43.7%	49.2%	0.35	45.3%	69.6%	54.9%	0.39	35.7%	87.9%	50.8%	0.23
GPT-J-6B	41.2%	30.6%	35.1%	0.17	35.5%	53.1%	42.6%	0.20	34.2%	85.1%	48.8%	0.19	41.2%	30.5%	35.0%	0.17	37.5%	56.3%	45.0%	0.24	35.5%	87.2%	50.5%	0.22
PolyCoder	41.6%	32.5%	36.5%	0.18	36.5%	56.9%	44.5%	0.23	34.2%	85.9%	49.0%	0.20	36.7%	29.0%	32.4%	0.13	35.3%	55.6%	43.2%	0.21	35.0%	87.0%	50.0%	0.22

(c) Self-attention-based methods.

Method	READING_first												CODING_first											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	44.9%	38.5%	41.5%	0.17	46.7%	75.9%	57.8%	0.39	41.7%	90.2%	57.0%	0.31	45.3%	38.0%	41.4%	0.18	45.1%	73.0%	55.7%	0.35	41.4%	89.5%	56.7%	0.30
CodeGen	9.8%	8.3%	9.0%	-0.17	26.0%	41.6%	32.0%	0.05	34.0%	84.1%	48.4%	0.19	9.5%	8.5%	9.0%	-0.17	25.6%	40.4%	31.3%	0.04	33.8%	83.8%	48.2%	0.19
CodeParrot	33.8%	27.5%	30.3%	0.10	37.9%	59.9%	46.4%	0.26	34.5%	86.2%	49.3%	0.20	44.0%	35.7%	39.4%	0.22	42.2%	67.2%	51.9%	0.35	34.6%	86.6%	49.4%	0.21
GPT-J-6B	6.5%	5.5%	6.0%	-0.21	23.0%	33.2%	27.2%	-0.03	33.0%	81.3%	46.9%	0.16	7.4%	6.2%	6.7%	-0.20	22.2%	33.0%	26.5%	-0.04	33.2%	81.9%	47.3%	0.17
PolyCoder	41.4%	32.5%	36.4%	0.19	41.7%	63.7%	50.4%	0.33	35.8%	87.9%	50.8%	0.23	43.8%	33.8%	38.1%	0.20	42.5%	64.8%	51.3%	0.34	35.8%	87.8%	50.9%	0.23
Method	READING_last												CODING_last											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	40.2%	31.9%	35.6%	0.12	41.2%	64.8%	50.4%	0.27	41.0%	88.8%	56.1%	0.29	41.9%	33.9%	37.5%	0.13	43.2%	69.6%	53.3%	0.31	41.4%	89.3%	56.6%	0.30
CodeGen	33.8%	24.8%	28.6%	0.08	39.5%	59.1%	47.4%	0.29	34.7%	85.2%	49.3%	0.21	31.0%	24.4%	27.3%	0.07	37.4%	57.1%	45.2%	0.25	35.1%	86.4%	49.9%	0.22
CodeParrot	60.1%	45.4%	51.7%	0.38	50.4%	76.4%	60.8%	0.48	37.0%	90.2%	52.4%	0.26	56.4%	43.2%	48.9%	0.35	48.5%	73.6%	58.4%	0.45	36.7%	89.7%	52.1%	0.26
GPT-J-6B	8.8%	7.0%	7.8%	-0.19	26.6%	42.7%	32.8%	0.05	33.4%	82.7%	47.5%	0.17	13.8%	10.6%	12.0%	-0.14	27.4%	41.8%	33.1%	0.06	33.6%	83.2%	47.9%	0.18
PolyCoder	23.1%	18.1%	20.3%	-0.03	34.3%	53.8%	41.9%	0.19	34.7%	86.4%	49.5%	0.21	32.8%	25.7%	28.9%	0.09	37.9%	58.8%	46.1%	0.26	35.4%	87.0%	50.3%	0.22
Method	READING_all												CODING_all											
	Top 5				Top 10				Top 20				Top 5				Top 10				Top 20			
	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA	P	R	F1	KA
Incoder	34.8%	32.5%	33.6%	0.08	39.0%	66.7%	49.2%	0.25	40.4%	88.2%	55.4%	0.28	40.6%	39.2%	39.9%	0.16	42.0%	71.8%	53.0%	0.31	41.2%	89.8%	56.5%	0.30
CodeGen	19.6%	14.7%	16.8%	-0.09	24.9%	37.0%	29.7%	-0.00	34.0%	83.9%	48.4%	0.19	28.4%	23.1%	25.4%	0.03	32.1%	50.7%	39.3%	0.15	34.5%	85.8%	49.2%	0.21
CodeParrot	41.8%	31.8%	36.1%	0.16	39.6%	61.3%	48.1%	0.28	35.1%	87.1%	50.1%	0.22	50.1%	39.7%	44.3%	0.28	43.4%	68.0%	53.0%	0.36	35.2%	87.3%	50.1%	0.22
GPT-J-6B	12.6%	10.2%	11.3%	-0.15	22.9%	35.0%	27.7%	-0.04	33.8%	83.6%	48.1%	0.18	30.6%	24.4%	27.2%	0.05	32.7%	51.2%	39.9%	0.15	34.6%	85.6%	49.2%	0.21
PolyCoder	28.7%	21.3%	24.4%	0.01	36.3%	56.3%	44.1%	0.21	34.8%	86.5%	49.6%	0.21	33.4%	25.7%	29.0%	0.08	38.4%	59.7%	46.8%	0.26	35.1%	86.9%	50.0%	0.22

Participants were then asked to rate each attention method by indicating their agreement with the following three statements on a 7-point Likert scale (1—completely disagree, 7—completely agree).

- Q1 *The model-attended keywords align with my attention when reading the natural language task description.*
- Q2 *This attention explains why the model succeeded in or failed at generating the correct code.*
- Q3 *I want to see this attention when working with code generation models in real life.*

The order of different attention calculation methods is randomized to mitigate the learning effect. We also do not reveal the names of the attention calculation methods to reduce bias. At the end of the user study, participants answer three open-ended questions about different attention calculation methods and the user study design. We ask these open-ended questions to study the correlation between model explainability and user trust. These questions include:

Q4 *Are you interested to know how the LLM generates the code?*
Q5 *What do you want to find out about the internal code generation process in LLM?*
Q6 *Do you trust this LLM? What do you need to know to improve the trust of the LLM?*

5 RESULTS

5.1 RQ1: To What Extent Is Model Attention Aligned With Human Attention?

To answer this question, we collect the top K keywords that the six LLMs attend to and compare them with the keywords labeled as important in the programmer attention dataset. As described in Section 4.3, we use San Martino's token overlapping metrics [22] and Krippendorff's alpha [49] to measure the attention alignment between LLMs and human programmers. When running models where the max output length can be set, we set the token limit to the number of tokens of the ground truth plus 20 to tolerate moderate redundancy.

Table 2a, Table 2b, and Table 2c present the attention alignment results when computing attention scores with *perturbation-based*, *gradient-based*, and *self-attention-based* methods, respectively. Because GPT-4 does not reveal its internal states during runtime, only perturbation-based methods are applicable. Therefore, this section discusses the results of the best perturbation-based method, *BERT_masking*. Section 5.3 compares different attention calculation methods in detail.

With *BERT_masking* method, from $K = 5$ to $K = 10$, the F1 scores of all models increase because as more and more tokens are selected by the models, we observe very high (around 90%) recall, which compensates for the declining precision. However, from $K = 10$ to $K = 20$, the F1 scores remain stable due to rapid declines in precision scores. On the other hand, the Krippendorff's alpha scores remain stable from $K = 5$ to $K = 10$ (except for GPT-4) but rapidly decrease from $K = 10$ to $K = 20$.

Overall, for all models and all K values, the Krippendorff's alpha does not change much and remains below 0.3, and the F1 scores remain below 0.6, indicating little agreement between model attention and human attention [58]. Among these models, GPT-4 showed the lowest attention overlap with human attention regarding both metrics. One possible explanation is that ultra-large models such as GPT-4 have developed a reasoning strategy different from that of human programmers. These results suggest a consistent attention misalignment between LLMs and programmers when generating code.

Finding 1

There is a consistent misalignment between LLM attention and programmer attention in all settings, indicating that LLMs do not reason programming tasks like human programmers.

5.2 RQ2: Can Attention Explain Errors of Code Generation Models?

To answer this question, the first two authors manually analyzed code generation errors made by the best two models in our study—GPT-4 and CodeGen-2.7B. In total, these two models generated 920 incorrect code solutions on the two benchmarks. We randomly sampled 211 incorrect solutions,

including 172 incorrect solutions from CodeGen-2.7B and 39 incorrect solutions from GPT-4. The sample size is statistically significant with a 90% confidence level and 5% margin of error.

The first two authors started with the first 50 code generation errors and independently checked whether the attention pattern of each code calculated by the *BERT_masking*, which gives the most aligned results in the quantitative experiments, could explain the error in it. For simple tasks, it takes about five minutes to check each one. For complicated tasks (e.g., tasks that require an understanding of specific math concepts), it takes around 10 to 15 minutes, since the authors need to manually debug the code and inspect its runtime values to understand the error first. After analyzing 50 errors, they discussed these errors with the other authors and summarized six common attention patterns that can be used to explain code generation errors:

- *Missing attention to critical conditions.* The task description mentions certain conditions or corner cases to handle. However, the model misses or incorrectly handles one or more such conditions since it does not attend to the words or phrases that describe the corresponding conditions.
- *Missing attention to important descriptive words of an operation or a data object.* The task description mentions an important property of an operation or a data object, such as "*largest* element" and "*ascending* order". However, the model does not attend to these descriptive words and thus generates a code solution with incorrect logic.
- *Missing attention to operation descriptions.* The task description mentions an operation or action, such as *open a file* and *sort a list*. However, the model fails to attend to the verb words or phrases. Therefore, the generated code performs the wrong action or does not perform the action.
- *Missing attention to data types.* Programming tasks often explicitly mention the expected type of inputs and outputs. Some task descriptions also mention the data type of some intermediate results. However, the model does not attend to some data type descriptions. This can lead to different types of errors, e.g., returning the wrong type of data, calling a method on the wrong type of object, etc.
- *Incorrect mapping between NL words and code elements.* In some cases, we observe the model attends to an important word or phrase correctly but the model maps it to a wrong method call, variable, parameter, value, or logic, potentially due to some conceptual misunderstanding of the semantics meaning of the NL words.

Then, they independently labeled the remaining errors, discussed their labeling with each other, and resolved the conflicts. In total, we found that errors in 57 of the 211 incorrect solutions (27%) can be explained by one of the five attention patterns mentioned above. Specifically, 54 of the 172 incorrect solutions from CodeGen-2.7B (31%) are explainable and 3 of the 39 incorrect solutions from GPT-4 (8%) are explainable. This finding suggests that neural attention analysis can be potentially applied to locate and repair a non-trivial portion of errors in LLM-generated code. Given that only 3 errors made by GPT-4 can be explained by attention misalignment patterns, this implies that weaker models such as CodeGen-2.7B are more likely to be affected by attention misalignment and thus their generation errors are more explainable. This is an interesting observation since GPT-4 also suffers from attention misalignment, as shown in Table 2, but its generation errors cannot be easily explained by attention analysis. This indicates that as the language models become large enough, they may have developed a different way of interpreting input prompts and generating content. We acknowledge that many errors cannot be easily explained by model attention. Such errors include syntax errors, undefined variable names, incorrect API usage, incorrect array index, infinite loop, etc. This result indicates that further analysis is required to understand the root causes of these errors.

We show the distribution and examples for each type of error below. In these examples, we highlight keywords with high attention scores from the model in blue.

5.2.1 Missing Attention to Critical Conditions. This prompt asks for a function that matches a string with a character *"a"* followed by zero or more character *"b"*s. However, CodeGen-2.7B generates a function that only matches words with one *"a"* followed by one or more *"b"*s. Our attention analysis shows conditional phrase *"followed by zero or more"* do not contain any of the most attended tokens, which explains why the model ignores the condition where the character *"a"* can be followed with no *"b"*s or more than one *"b"*. 6 incorrect solutions fall into this category.

```
# Write a function that matches a string that has an ``a'' followed by zero or more ``b'' s.
def function(string):
    return string.count('ab') >= 1
```

5.2.2 Missing Attention to Important Descriptive Words of an Operation or a Data Object. The prompt below asks for a function that calculates the n^{th} power of all elements in a list. However, the code generated by CodeGen-2.7B calculates the n^{th} power of a specific element without using the lambda function. Our attention analysis shows the property phrase *"individual items"* contains none of the most attended words in the prompt. This explains why the model fails to perform the requested operation on all items instead of a specific item. 13 incorrect solutions fall into this category.

```
# Write a function to find the n-th power of individual elements in a list using lambda function.
def function(list, n):
    return list[n-1] ** n
```

5.2.3 Missing Attention to Operation Descriptions. The prompt below asks for a function that counts how many pairs of elements at the same index in two given arrays are equal. However, CodeGen-2.7B generates a function that checks whether the two strings are equal. Our attention analysis shows the model the operator keyword *"count"* is not one of the most attended words in the prompt, which explains why the generated code does not perform counting as requested. 6 incorrect solutions fall into this category.

```
# Write a function to count the same pair in two given lists using map function.
def function(x,y):
    return x == y
```

5.2.4 Missing Attention to Data Types. This prompt asks for a function that matches a word that contains *"z"* but does not have *"z"* at the start or end. However, CodeGen-2.7B generates a function that checks whether the string retrieved from removing the first character from the given word equals the string retrieved from removing the last character from the given word. Our attention analysis shows that the data type keyword *"z"* is not one of the most attended words in the prompt. This explains why the generated code fails to check for the existence of a specific character in the given word. 7 errors belong to this category.

```
# Write a function that matches a word containing 'z', not at the start or end of the word.
def function(word):
    return word[1:] == word[:-1]
```

5.2.5 Incorrect Mapping Between NL Words and Code Elements. This prompt asks for a function that calculates the n^{th} bell number. However, instead of invoking the correct formula for bell number, CodeGen-2.7B calculates the factorial of n. Our attention analysis shows the keyword *"bell"* is one of the most attended words in the prompt. However, it still generates incorrectly for this prompt because it does not understand the semantic meaning of *"bell number"*. 25 incorrect solutions belong to this category.

```
# Write a python function to find nth bell number.
def function(n):
    if n == 1:
        return 1
    else:
        return n * function(n - 1)
```

Finding 2

Attention analysis can explain 27% of generation errors in a benchmark of 211 errors. These errors provide insights into the limitations of LLMs.

5.3 RQ3: What Is the Impact of Attention Calculation Methods on the Alignment?

To compare different attention calculation methods, we calculated the average San Martino's F-1 score and Krippendorff's alpha of all five models (except for GPT-4) for all K (Table 3). Table 3 shows that *BERT_masking* gives the highest alignment in both metrics for $K = 5, 10$. However, the other perturbation-based method, *SHAP*, does not outperform other methods as *BERT_masking* did. On the other hand, gradient-based methods are generally better than self-attention-based methods, especially for $K = 5$ in terms of both metrics. Surprisingly, attention scores computed by self-attention-based methods (e.g., *CODING_first*) are least aligned with human attention, especially for smaller K values. One possible reason is that other computation units in the transformer architecture, such as the feed-forward layers, also play an important role in the code generation process. For instance, Geva et al. find that feed-forward layers influence model predictions by promoting concepts in the vocabulary space [35]. Thus, only considering self-attention layers does not fully capture the influence of each input token on model predictions.

Furthermore, within gradient-based methods, we observe that calculating attention distribution during the coding stage (e.g., *Saliency_coding*) gives better alignment than the attention distribution during the reading stage (e.g., *Saliency_reading*). This pattern can be observed for all K values (i.e., $K = 5, 10$). However, the differences between *Input×Gradient* and *Saliency* are trivial.

Table 3. The average San Martino's F-1 score and Krippendorff's Alpha over all five models (excluding GPT-4) in various K settings. The highest score in each column is highlighted in yellow.

Method	Top 5		Top 10		Top 20	
	F1	KA	F1	KA	F1	KA
BERT_perturbation	45.7%	0.27	51.4%	0.26	52.1%	0.12
SHAP	30.6%	0.11	41.5%	0.18	48.0%	0.16
Input×Gradient_reading	38.8%	0.20	49.6%	0.30	51.1%	0.23
Input×Gradient_coding	39.4%	0.21	50.3%	0.31	52.0%	0.25
Saliency_reading	38.2%	0.19	48.1%	0.27	50.8%	0.23
Saliency_coding	38.4%	0.19	48.8%	0.29	51.8%	0.24
READING_first	24.6%	0.01	42.7%	0.20	50.5%	0.22
READING_last	28.8%	0.07	46.6%	0.26	51.0%	0.23
READING_all	24.4%	0.00	39.8%	0.14	50.3%	0.22
CODING_first	26.9%	0.05	43.4%	0.21	50.5%	0.22
CODING_last	30.9%	0.10	47.2%	0.27	51.4%	0.24
CODING_all	33.2%	0.12	46.4%	0.25	51.0%	0.23

Finally, as discussed in Section 2.3.1, there is no consensus on how to aggregate self-attention scores [10, 97, 99]. We experimented with six different self-attention aggregation methods used in previous work to find the best self-attention aggregation method. First, among the three options—(1) only summing attention scores in the first layer, (2) only summing attention scores in the last layer, and (3) summing attention scores from all layers—only summing attention scores in the last layer produces the attention distributions that are most aligned with human attention distributions in both reading and coding stages, except when K is set to 5 in the coding stage. This suggests that future research should consider the last layer when leveraging self-attentions to interpret code language models. Similar to the finding in gradient-based methods, for self-attention-based methods, attention distributions at the coding

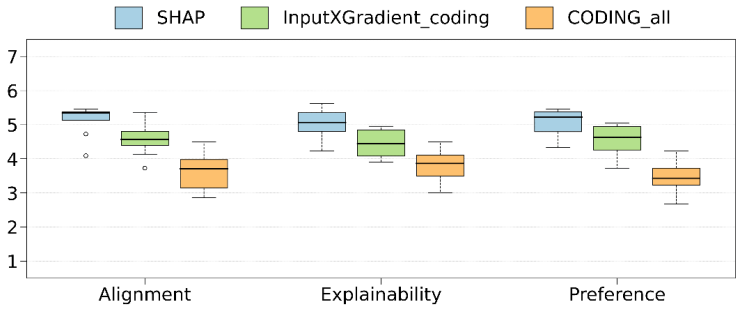

Fig. 5. Participants' choices over different attention calculation methods in three dimensions.

stage (e.g., *CODING_first*) are consistently more aligned with human attention than those at the reading stage (e.g., *READING_first*).

Finding 3

BERT_masking produces the best attention alignment to human programmers among all methods. Gradient-based methods are generally better than self-attention-based methods. For gradient-based and self-attention-based methods, attention distributions in the coding stage produce higher alignment. For self-attention-based methods, attention distributions in the last layer produce the highest alignment.

5.4 RQ4: Which Attention Calculation Method Is the Most Preferred?

Figure 5 shows participants' assessment about *SHAP*, *Input×Gradient_coding*, and *CODING_all* in terms of the alignment with their own attention, the explainability of the computed attention, and their own preference (Q1-Q3 in Section 4.4). Overall, the perturbation-based method, *SHAP*, is favored over the other two methods in all three aspects. The average rating on the attention alignment of *SHAP* is 5.13, while the average for *InputXGradient_coding* and *CODING_all* are 4.59 and 3.62, respectively.

Furthermore, participants suggested that *SHAP* has better explainability than *InputXGradient_coding* (mean difference: 0.59, Welch's t-test, $p = 0.02$) and *CODING_all* (mean difference: 1.26, Welch's t-test, $p = 0.00001$). However, according to participants' responses about their trust in LLMs (Q6 in Section 4.4), 14 out of 22 participants expressed a lack of confidence and trust, even after seeing the attention-based explanations. For example, P3 wrote, *"I want to know whether the LLM model can provide the reasons why they generate the code. For instance, the reference code."* Participants also asked for more fine-grained attention analysis. Specifically, they wished to see which parts of the input are responsible for generating which parts of the output. For instance, P10 said, *"[I want to know] how the LLM determines the input and output, what are the discriminative words that drive different generations."* Participants also showed more preference for *SHAP* over the other two methods. The mean preference differences between *SHAP* and *InputXGradient_coding* and between *SHAP* and *CODING_all* is 5.04 vs. 4.56 (Welch's t-test, $p = 0.05$) and 5.04 vs. 3.48 (Welch's t-test, $p = 0.00009$).

Finding 4

Overall, participants preferred the perturbation-based method over the gradient-based and the self-attention-based methods. However, participants still felt a lack of trust in LLMs after seeing the attention-based explanations and wished to see richer explanations such as reference code and fine-grained attention mapping between text and code.

6 IMPLICATIONS AND OPPORTUNITIES

Our study has several significant implications that benefit the development of more reliable and more accurate LLMs for code generation in the future.

In RQ1, we discovered a consistent misalignment between human and model attention in all six models. While it is possible that LLMs use an entirely different method to reason about task descriptions compared with human programmers, it is arguable whether such a method is indeed good for programmers due to concerns about interpretability, robustness, and trust. Many studies have shown that human-aligned models are perceived as more trustworthy by humans [8, 34, 38, 42, 46, 80]. Furthermore, in practice, LLMs today can reliably solve a limited number of simple programming tasks and struggle to handle more sophisticated or custom tasks, which indicates a huge space for improvement. Thus, we believe investigating the attention patterns of LLMs is a worthwhile effort to help us understand how LLMs generate code and why LLMs make some mistakes and also inform new opportunities to improve LLMs.

In RQ2, we manually analyzed the attention of 211 incorrect generations of the best two models in our paper (GPT-4 and CodeGen-2.7B) and found 27% of the errors can be explained by incorrect attention alignment. Our finding suggests the potential of fixing these generation errors by adjusting the attention alignment. Similarly, previous work in Computer Vision has shown that the performance of neural models can be improved if we force their attention to align with humans [31, 44, 59, 63]. One potential solution is to extend the loss function of LLMs with a penalty term that measures the KL divergence between model attention and human attention. During training, the loss will increase when model attention deviates from human attention. As a result, the LLM will be trained to distribute attention in a way similar to human programmers. We have open-sourced our human attention dataset to facilitate future work on attention alignment.

Furthermore, the model attention patterns in RQ3 can help to improve the robustness of code generation models by developing new attention-based adversarial training methods. Most adversarial training methods only apply small perturbations to random tokens in a prompt, without considering the importance of a token to the model [11, 79, 92, 98]. Our results show that it is worthwhile to prioritize important tokens during perturbation. Perturbing these important words and asking the model to generate code for these perturbed prompts may reveal robustness issues more efficiently.

In RQ3, we compared 12 attention calculation methods with quantitative experiments (Table 2). Therefore, our study provides practical guidelines for choosing the attention calculation method for future research on LLM-based code generation models. Our experiment results indicate developers who want to measure the attention of code generation models should first consider *BERT_masking* since it consistently achieves the best alignment with human attention in all except for two settings (Table 3). Between self-attention-based and gradient-based methods, researchers should prioritize gradient-based methods that generally give better and more stable alignment in most settings (Table 3). For gradient-based and self-attention-based methods, calculating the attention distribution of the coding stage gives better alignment (Table 3). Finally, when leveraging self-attention to

interpret transformer-based code generation models, researchers should consider using the self-attention scores computed from the last layer, which is demonstrated to be more aligned with human attention (Table 3).

In RQ4, we conducted a user study to compare developers' perceptions on different explanation methods. Most participants considered SHAP as the best XAI method for LLM code generation models. This finding suggests that future researchers may want to use the perturbation-based method as the default XAI method for LLM-based code generation. Furthermore, in the post-study survey, participants also asked for more fine-grained attention analysis, such as revealing the association between individual input and output tokens. This highlights the need for new XAI methods to interpret LLM-based code generation models.

In the future, we would also like to explore how to teach humans how LLMs interpret code. Understanding how LLMs interpret code can inspire human programmers to craft prompts that are more understandable by LLMs and thus guide LLMs to generate better code.

7 THREAT TO VALIDITY

Internal validity. One potential threat lies in the manual labeling process. Two programmers manually labeled the important words to understand a task description and implement the correct function. Since this criterion of keywords is highly subjective, the words selected by the two labelers may not represent the choice of a large pool of programmers. To mitigate this threat, the authors established a labeling standard through discussion and achieved substantial agreement on the labeling. Furthermore, we invite a third labeler to validate our annotated dataset by independently labeling the 164 prompts from the HumanEval dataset. We calculated the Fleiss' Kappa score among the labels of three labelers and the result (0.64) indicates a significant agreement.

External validity. One potential threat to external validity is that we have only experimented with one programming language, Python. We cannot guarantee that our findings generalize to another language.

Construct validity. One potential threat to construct validity lies in the survey design. As we know, programming problems are mentally demanding to solve and programmers may have different views on which part of the prompt should the model attend to. However, in the current design, only 22 participants were involved. The small sample size may result in findings that are not generalizable. To combat this threat, we only invited participants who are experienced in Python programming and have used code generation LLMs at least once to control the quality of the user study.

8 RELATED WORK

8.1 Code Generation From Natural Language

Since CodeBERT [28], there has been a large body of literature where Large Language Models (LLMs) are used in code generation [5, 27, 29, 32, 37, 62, 66, 74, 82, 86, 88, 91, 96]. Both Guo et al. [37] and Zeng et al. [96] used a pre-trained BERT [25] model to encode NL questions and database schemas for text-to-SQL generation. CodeBERT adopts the same model architecture as BERT but is trained with a hybrid objective on code and text data from GitHub repositories [28]. Codex, CodeGPT, and GraphCodeBERT improve CodeBERT by leveraging data flow in the pre-training stage [36]. Recently, modern LLMs such as GPT-4 and Google Bard excel in code generation tasks on various benchmarks [14, 24]. The highly accurate and contextually rich code snippets generated by these models enable the automation of various programming tasks.

In addition to developing new LLMs for code, prior work has also presented new methods to enhance LLMs for more accurate and robust code generation [15, 16, 51, 66, 74, 94, 100]. Instead of

directly generating code from text, REDCODER [66] first retrieves similar code snippets from a corpus and then passes them along with the text description to an LLM for code generation. Shen et al. [74] proposed to leverage domain knowledge from documentation and code comments to assist code generation. Chakraborty et al. proposed a new pre-training task called *naturalizing of source code* (i.e., translate an artificially created code to a human-written form) to help an LLM learn how to generate natural code [15]. Chen et al. proposed to use an LLM to automatically generate test cases to examine the code generated by the same LLM [16]. Zan et al. proposed to first decompose a LLM to two LLMs, one for generating program sketches and the other for filling the sketches [94]. SkCoder [51] is designed to mimic developers' code reuse behavior by first retrieving a code example related to a given task, extracting a sketch from it, and editing the sketch based on the task description.

8.2 Empirical Studies on Code Generation

Recently, many studies have evaluated LLM-based code generation models in different aspects, including performance [27, 39, 55, 71, 97], robustness [56, 103], security [6, 67, 93], code smells [77], usability [9, 12, 83, 91], and licensing [19]. The most related to us are those that investigate the explainability of LLMs for code [45, 53, 85, 99]. Karmakar et al. studied what pre-trained BERT-based models such as CodeBERT and GraphCodeBERT have learned about code using probing tasks [45]. A probing task is essentially a prediction task on a specific code property, such as code length and cyclomatic complexity, to test whether the model is sensitive to the property. Compared with our work, they did not analyze the code generation process, e.g., why and how certain code is generated based on a text description. Liguori et al. proposed a perturbation-based method to evaluate NMT models for code generation [53]. In addition, Zhang et al. investigated the code generation process by analyzing the self-attention layers in CodeBERT and GraphCodeBERT [99]. Wan et al. did a similar self-attention analysis and also designed a new probe task on code structures to analyze whether LLMs have learned information about code structures [85]. Compared with these studies, our work differs by analyzing the consistency between LLMs' attention and programmers' attention on the NL description of a programming task.

8.3 Model Attention Analysis in Other Domains

Many attention calculation methods have been developed to explain models in other domains. For example, in CV domain, Selvaraju et al. [73] proposed to use the gradients of a convolutional neural network (CNN) to indicate the importance of each pixel in an image for a specific prediction. Similarly, Zhou et al. [102] used the global average pooling mechanism to calculate the importance of each region for CNN to classify an image correctly. Autonomous driving is another domain where the need for explainability is strong. For example, Zeiler et al. [95] use deconvolution layers to understand how autonomous vehicles capture real-time image segments using CNNs. In another work, Chen et al. [17] proposed a data-efficient policy learning approach called Semantic Predictive Control (SPC) that explains how perceived environmental states are mapped to actions.

9 CONCLUSION

This paper presents an empirical study on attention alignment between LLM-based code generation models and human programmers. Our results reveal that there is a consistent misalignment between LLMs' and programmers' attention. Among the 12 attention calculation methods, perturbation-based methods produced attention scores that were better aligned with human attention and were also more preferred by user study participants. Based on our study results, we further discuss several implications and future research opportunities for better interpretation and performance improvement of LLM-based code generation models.

10 DATA AVAILABILITY

Our code and data are available on a GitHub repository [47].

References

[1] 2022. CodeParrot. https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot.

[2] 2023. ChatGPT. http://chat.openai.com.

[3] 2023. GPT-4 Parameters: Unlimited guide NLP's Game-Changer. https://medium.com/@mlubbad/the-ultimate-guide-to-gpt-4-parameters-everything-you-need-to-know-about-nlps-game-changer-109b8767855a.

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[5] Alex Andonian, Quentin Anthony, et al. 2021. *GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch.* https://doi.org/10.5281/zenodo.5879544

[6] Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741* (2022).

[7] Jacob Austin, Augustus Odena, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[8] Aakash Bansal, Bonita Sharif, and Collin McMillan. 2023. Towards Modeling Human Attention from Eye Movements for Neural Source Code Summarization. *Proceedings of the ACM on Human-Computer Interaction* 7, ETRA (2023), 1–19.

[9] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[10] Joshua Bensemann et al. 2022. Eye gaze and self-attention: How humans and transformers attend words in sentences. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics*. 75–87.

[11] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*. PMLR, 896–907.

[12] Christian Bird et al. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[13] Angie Boggust, Benjamin Hoover, Arvind Satyanarayan, and Hendrik Strobelt. 2022. Shared interest: Measuring human-ai alignment to identify recurring patterns in model behavior. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–17.

[14] Sébastien Bubeck et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[15] Saikat Chakraborty et al. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

[16] Bei Chen et al. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[17] Jianyu Chen, Shengbo Eben Li, and Masayoshi Tomizuka. 2021. Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems* 23, 6 (2021), 5068–5078.

[18] Wenhu Chen, Evgeny Matusov, Shahram Khadivi, and Jan-Thorsten Peter. 2016. Guided alignment training for topic-aware neural machine translation. *arXiv preprint arXiv:1607.01628* (2016).

[19] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 167–178.

[20] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What Does BERT Look at? An Analysis of BERT's Attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. 276–286.

[21] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[22] Giovanni Da San Martino et al. 2019. Fine-grained analysis of propaganda in news article. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, 5636–5646.

[23] Misha Denil, Alban Demiraj, and Nando De Freitas. 2014. Extraction of salient sentences from labelled documents. *arXiv preprint arXiv:1412.6815* (2014).

[24] Giuseppe Destefanis, Silvia Bartolucci, and Marco Ortu. 2023. A Preliminary Analysis on the Code Generation Capabilities of GPT-3.5 and Bard AI Models for Java Functions. *arXiv preprint arXiv:2305.09402* (2023).
[25] Jacob Devlin et al. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*. 4171–4186.
[26] Ahmed Elnaggar et al. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).
[27] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]
[28] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.
[29] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Language Processing. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 307–316.
[30] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.
[31] Ruth C Fong, Walter J Scheirer, and David D Cox. 2018. Using human brain activity to guide machine learning. *Scientific reports* 8, 1 (2018), 5397.
[32] Daniel Fried et al. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL
[33] Andrea Galassi, Marco Lippi, and Paolo Torroni. 2020. Attention in natural language processing. *IEEE transactions on neural networks and learning systems* 32, 10 (2020), 4291–4308.
[34] Yuyang Gao et al. 2022. Aligning eyes between humans and deep neural network through interactive attention alignment. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–28.
[35] Mor Geva, Avi Caciularu, Kevin Ro Wang, and Yoav Goldberg. 2022. Transformer feed-forward layers build predictions by promoting concepts in the vocabulary space. *arXiv preprint arXiv:2203.14680* (2022).
[36] Daya Guo et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[37] Jiaqi Guo et al. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4524–4535.
[38] Christopher Hazard et al. 2022. Importance is in your attention: agent importance prediction for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2532–2535.
[39] Dan Hendrycks et al. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).
[40] Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. 2020. Aligning AI with Shared Human Values. *arXiv preprint arXiv:2008.02275* (2020).
[41] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. 2018. Evaluating feature importance estimates. (2018).
[42] Siteng Huang, Min Zhang, Yachen Kang, and Donglin Wang. 2021. Attributes-guided and pure-visual attention alignment for few-shot recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 7840–7847.
[43] Paras Jain and Ajay Jain. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.
[44] Shaohua Jia et al. 2018. Biometric recognition through eye movements using a recurrent neural network. In *2018 IEEE International Conference on Big Knowledge (ICBK)*. IEEE, 57–64.
[45] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1332–1336.
[46] Iuliia Kotseruba, Amir Rasouli, and John K Tsotsos. 2016. Joint attention in autonomous driving (JAAD). *arXiv preprint arXiv:1609.04741* (2016).
[47] Bonan Kou. 2024. Attention-Alignment-Empirical-Study. https://github.com/BonanKou/Attention-Alignment-Empirical-Study.
[48] Klaus Krippendorff. 2004. Reliability in content analysis: Some common misconceptions and recommendations. *Human communication research* 30, 3 (2004), 411–433.
[49] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.
[50] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
[51] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. *arXiv preprint arXiv:2302.06144* (2023).
[52] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding neural networks through representation erasure. *arXiv preprint arXiv:1612.08220* (2016).
[53] Pietro Liguori et al. 2022. Can NMT understand me? towards perturbation-based evaluation of NMT models for code generation. In *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*.

IEEE, 59–66.
[54] Shusen Liu et al. 2018. Nlize: A perturbation-driven visual interrogation tool for analyzing and interpreting natural language inference models. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 651–660.
[55] Xiaodong Liu, Ying Xia, and David Lo. 2020. An Empirical Study on the Usage of Transformer Models for Code Completion. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 408–418.
[56] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2023. On the Reliability and Explainability of Automated Code Generation Approaches. *arXiv preprint arXiv:2302.09587* (2023).
[57] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).
[58] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[59] Cristina Melício et al. 2018. Object detection and localization with artificial foveal visual attention. In *2018 Joint IEEE 8th international conference on development and learning and epigenetic robotics (ICDL-EpiRob)*. IEEE, 101–106.
[60] Christoph Molnar. 2020. *Interpretable machine learning*. Lulu. com.
[61] Ernst Niebur. 2007. Saliency map. *Scholarpedia* 2, 8 (2007), 2675.
[62] Erik Nijkamp, Bo Pang, et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.
[63] Afonso Nunes, Rui Figueiredo, and Plinio Moreno. 2020. Learning to search for objects in images from human gaze sequences. In *Image Analysis and Recognition: 17th International Conference*. Springer, 280–292.
[64] Matteo Paltenghi and Michael Pradel. 2021. Thinking like a developer? comparing the attention of humans with neural models of code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 867–879.
[65] Kishore Papineni et al. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[66] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2719–2734.
[67] Hammond Pearce et al. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[68] Md Rafiqul Islam Rabin, Vincent J Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 441–452.
[69] Shuo Ren et al. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[70] Marco Tulio Ribeiro et al. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[71] Rafael R Rodrigues et al. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 327–336.
[72] Jaydeb Sarker, Sayma Sultana, Steven R Wilson, and Amiangshu Bosu. 2023. ToxiSpanSE: An Explainable Toxicity Detection in Code Review Comments. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12.
[73] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.
[74] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1533–1543.
[75] Donghee Shin. 2021. The effects of explainability and causability on perception, trust, and acceptance: Implications for explainable AI. *International Journal of Human-Computer Studies* 146 (2021), 102551.
[76] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International conference on machine learning*. PMLR, 3145–3153.
[77] Mohammed Latif Siddiq et al. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.
[78] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[79] Shashank Srikant et al. 2020. Generating Adversarial Computer Programs using Optimized Obfuscations. In *International Conference on Learning Representations*.
[80] Andrea Stocco et al. 2022. Thirdeye: Attention maps for safe autonomous driving systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[81] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International conference on machine learning*. PMLR, 3319–3328.
[82] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, Incorporated.
[83] Priyan Vaithilingam et al. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI conference on human factors in computing systems extended abstracts*. 1–7.
[84] Shikhar Vashishth, Shyam Upadhyay, Gaurav Singh Tomar, and Manaal Faruqui. 2019. Attention interpretability across nlp tasks. *arXiv preprint arXiv:1909.11218* (2019).
[85] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.
[86] Bailin Wang et al. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7567–7578.
[87] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.
[88] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.
[89] Katharina Weitz et al. 2019. " Do you trust me?" Increasing user-trust by integrating virtual agents in explainable AI interaction design. In *Proceedings of the 19th ACM International Conference on Intelligent Virtual Agents*. 7–9.
[90] Zhiyong Wu, Yun Chen, Ben Kao, and Qun Liu. 2020. Perturbed Masking: Parameter-free Probing for Analyzing and Interpreting BERT. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4166–4176.
[91] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
[92] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[93] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 62–71.
[94] Daoguang Zan, Bei Chen, et al. 2022. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 2369–2375. https://doi.org/10.24963/ijcai.2022/329
[95] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, 818–833.
[96] Yu Zeng et al. 2020. RECPARSER: A Recursive Semantic Parsing Framework for Text-to-SQL Task.. In *IJCAI*. 3644–3650.
[97] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.
[98] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1169–1176.
[99] Kechi Zhang, Ge Li, and Zhi Jin. 2022. What does Transformer learn about source code? *arXiv preprint arXiv:2207.08466* (2022).
[100] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1073–1084.
[101] Haiying Zhao, Wei Zhou, Xiaogang Hou, and Hui Zhu. 2020. Double attention for multi-label image classification. *IEEE Access* 8 (2020), 225539–225550.
[102] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2921–2929.

[103] Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. 2023. On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. 1090–1102.

Received 2023-09-28; accepted 2024-04-16