Follow-Up Attention: An Empirical Study of Developer and Neural Model Code Exploration

Matteo Paltenghi ©, Rahul Pandita ©, Austin Z. Henley ©, and Albert Ziegler ©

Abstract—Recent neural models of code, such as OpenAI Codex and AlphaCode, have demonstrated remarkable proficiency at code generation due to the underlying attention mechanism. However, it often remains unclear how the models actually process code, and to what extent their reasoning and the way their attention mechanism scans the code matches the patterns of developers. A poor understanding of the model reasoning process limits the way in which current neural models are leveraged today, so far mostly for their raw prediction. To fill this gap, this work studies how the processed attention signal of three open large language models - CodeGen, InCoder and GPT-J - agrees with how developers look at and explore code when each answers the same sensemaking questions about code. Furthermore, we contribute an open-source eye-tracking dataset comprising 92 manually-labeled sessions from 25 developers engaged in sensemaking tasks. We empirically evaluate five heuristics that do not use the attention and ten attention-based post-processing approaches of the attention signal of CodeGen against our ground truth of developers exploring code, including the novel concept of follow-up attention which exhibits the highest agreement between model and human attention. Our follow-up attention method can predict the next line a developer will look at with 47% accuracy. This outperforms the baseline prediction accuracy of 42.3%, which uses the session history of other developers to recommend the next line. These results demonstrate the potential of leveraging the attention signal of pre-trained models for effective code exploration.

Index Terms—Attention mechanism, code exploration, eye tracking, follow-up attention, neural models of code.

I. INTRODUCTION

LARGE language models (LLMs) pre-trained on code such as Codex [1], CodeGen [2], and AlphaCode [3] have demonstrated remarkable proficiency at program synthesis and competitive programming tasks. Yet our understanding of why they produce a particular solution is limited. In large-scale practical applications, the models are often used for their prediction alone, i.e., as generative models, and the way they reason about code internally largely remains untapped.

These models are often based on the attention mechanism [4], a key component of the transformer architecture [5]. Besides providing substantial performance benefits, attention weights have been used to provide interpretability of neural models [6], [7], [8]. Additionally, existing work [9], [10], [11], [12] also suggests that the attention mechanism reflects or encodes objective properties of the source code processed by the model. We argue that just as software developers consider different locations in the code individually and follow meaningful connections between them, the self-attention of transformers connects and creates information flow between similar and linked code locations. This raises a question:

Are human attention and model attention comparable? And if so, can the knowledge about source code conveyed by the attention weights of neural models be leveraged to support code exploration?

Although there are other observable signals that might capture the concept of relevance, such as gradients-based [13], [14] or layer-wise relevance propagation [15], this work focuses on approaches using only the attention signal. The reasons for this choice are two: (1) almost all state-of-the-art models of code are based on the transformer block [5], and the attention mechanism is ultimately its fundamental component, so we expect the corresponding attention weights to carry directly meaningful information about the models’ decision process; (2) attention weights can be extracted almost for free during the generation with little runtime overhead since the attention is computed automatically during a single forward pass.

Answering the main question of this study requires a dataset tracking developers’ attention. In this work, we use visual attention as a proxy for the elements to which developers are paying mental attention while looking at code. However, the existing datasets of visual attention are not suitable for our purposes. Indeed, they either put the developers in an unnatural, and thus possibly biasing, environment where most of the vision is blurred [8], requiring participants to move the mouse over tokens to reveal them, or they contain few and very specific code comprehension tasks [16] on code snippets too short to exhibit any interesting code navigation pattern. This blurring method can introduce bias by forcing unnatural interactions, potentially affecting how developers naturally explore and understand code. To address these limitations and stimulate...
developers to not only glance at code, but also to deeply reason about it, we prepare an ad-hoc code understanding assignment called the sensemaking task. This involves questions on code, including mental code execution, side-effects detection, algorithmic complexity, and deadlock detection. Moreover, using eye-tracking, we collect and share a dataset of 92 valid sessions with developers.

On the neural model side, motivated by some recent successful applications of few-shot learning in code generation and code summarization [17], [18] and even zero-shot in program repair [19], the sensemaking task is designed to be a zero-shot task for the model with a specific prompt that triggers it to reason about the question at hand. Then we query three LLMs of code, namely CodeGen [2], InCoder [20] and GPT-J [21] on the same sensemaking task and compare their attention signal\(^1\) to the attention of developers. The correlation with CodeGen, the largest model, is the highest among the LLMs studied (r=+0.23), motivating the use of raw and processed versions of CodeGen’s attention signal for code exploration. To that end, we experimentally evaluate how well existing and novel attention post-processing methods align with the code exploration patterns derived from our dataset’s chronological sequence of eye-fixation events. To the best of our knowledge, this work is the first to investigate the attention signal of these pre-trained models to support code exploration, a specific code-related task, directly related to code reading work [22], [23].

We empirically demonstrate that post-processing methods based on the attention signal can be well aligned with the way developers explore code. In particular, using the novel concept of follow-up attention, we achieve the highest overlap with the developers’ ground truth on which line to explore next.

**Contributions:** This paper makes the following contributions:

- **Sensemaking Task** A novel task and setup to deepen our understanding of how the LLM attention connects to the temporal sequence of location shifts regarding developer focus.
- **Eye-Tracking Dataset** A novel dataset of 92 eye-tracking sessions of 25 developers engaged in sensemaking tasks while using a common code editor with code written in three popular programming languages (Python, C++, and C#).
- **Follow-up Attention** The analytical formula for follow-up attention, a novel post-processing approach derived solely from the attention signal, which aligns well with the developer interaction of which line to look at next when exploring code.
- **Empirical Study** The first comparison of both effectiveness and visual attention of LLMs and developers when reasoning on sensemaking questions. An empirical evaluation comprising ten post-processing approaches of the attention signal, five heuristics, and an ablation study of the follow-up attention against the collected ground truth of developers exploring code.

---

1Attention signal refers to the attention weights produced during a forward pass by the transformer blocks.

---

II. RELATED WORK

This section provides an overview of related work around the explanatory role of attention and previous studies of the attention of neural models and developers when reasoning on code.

A. Attention as Explanation

Initially, preliminary work [24] studying attention weights of recurrent neural models has found that the attention weights do not always agree with other explanation methods and that alternative weights can be adversarially constructed while still preserving the same model prediction. However, in response, Wiegreffe and Pinter [25] have shown how the alternative attention weights can be constructed only per a single instance prediction, whereas obtaining a model which is consistently wrong in its explanations is very unlikely to happen. On the same line, Tutek and Šnajder [26] have proposed four regularization methods to mitigate the adversarial exploitation of attention weights for recurrent models, including the use of residual connections which are natively embedded into transformers [5], the building blocks of the LLMs studied in this work. To further corroborate this connection between attention and explanation, Rabin et al. [27] have shown how even Sivand, an explainability technique based on program simplification, pinpoint important tokens which largely overlap with those reported by the attention mechanism.

B. Attention Studies of Neural Models of Code

Paltenghi and Pradel [8] have compared the attention weights of neural models of code and developers’ visual attention when performing a code summarization task, and found a strong positive correlation on the copy attention mechanism for an instance of a pointer network [28]. Further works [9], [11] have then shown how the attention weights of pre-trained models on source code capture important properties of the abstract syntax tree of the program. However, none of them considered the use of the attention signal for a code-related task, such as code exploration. Moreover, they are limited to relatively small self-attention transformer models, whereas we study the attention of CodeGen [2], InCoder [20] and GPT-J [21], large generative models with masked self-attention.

C. Eye-Tracking Studies

Turner et al. [29] conducted an eye-tracking study involving 38 students fixing or describing five simple Python and C++ programs (5-13 LoC) showing that the fixation duration is comparable between the two languages. Beelders [30] has qualitatively observed the eye movement of 36 students and four lecturers when reading and mentally executing a short C# program (12 LoC). An eye-tracking dataset with 216 participants has been collected by [16], however, they only consider two short snippets (11-22 LoC) of code, since they do not support scrolling. Similarly, Blascheck and Sharif [22] and Busjahn et al. [23] have studied the reading order in C++ and Java code.
comprehension task focusing on six small programs that could fit into a single screen, whereas we consider longer snippets and a much larger dataset of 45 unique tasks. Sharifi et al. [31] have recently studied code navigation strategies on Java code with eye tracking involving 36 participants focusing on the bug fixing process, however, we study the sensemaking task which might elicit a different kind of reasoning compared to bug-fixing. To more closely mimic real-world setups in integrated development environments (IDEs), Guarnera et al. [32] propose iTrace, an eye-tracking plugin for IDEs that can track developers’ eye movements in more realistic and dynamic coding environments beyond a single screen of code. Further studies, including Fakhoury et al. [33], have proposed Gazel, an IDE plugin that supports eye tracking in the context of source code editing. Following this latest trend, we also use an IDE plugin to collect the eye-tracking data, allowing for a more realistic coding environment.

III. SENSEMAKING TASK

To study developers’ and models’ attention, we prepare a code understanding task called sensemaking task because the developer has to “make sense” of code to answer the question correctly. One sensemaking task is contained in a single source code file composed of four sections: (1) a brief description of the context of the main code snippet (e.g., The following code reasons about triangles in the geometrical sense.), (2) the main code snippet, either sourced on the internet or written from scratch by the authors, (3) a sensemaking question to stimulate the reasoning (i.e., Question:), and (4) a final prompt to trigger the model’s answer (i.e., Answer:). Note that all the sections except the main snippet are in the form of code comments. Fig. 1 shows an example task, whereas the full list of questions can be seen in the Table I.

To source the tasks for our study, we rely on GeeksforGeeks, a well-known website for programming education and practice. This website offers a variety of problem statements that are commonly used in typical technical interviews by modern software companies, as shown by previous research [34]. Therefore, we expect that the software developers would have some familiarity with the type of these programs. We then create specific sense-making questions about these programs, inspired by the kind of questions that an interviewer might pose, such as asking about the output, complexity, correctness, or code modification. Indeed, many of our questions are concrete instances of question templates such as “What is the purpose of the code?” (nqueens_Q1), “What is the program supposed to do?” (tree_Q3) or “What code could have caused this behavior?” (triangle_Q1), which also have been identified as questions that software engineers often ask themselves in a real working setting [35]. To stimulate code exploration, many of them are also instances of reachability questions [36]; namely, they involve the search over all feasible paths of a program to locate target statements matching search criteria. Some examples of these are “What are the implications of this change?” (triangle_Q3) or “How does application behavior vary in these different situations that might occur?” (triangle_Q2, tree_Q1, multithread_Q3). We prepare five main snippets and create three unique questions for each of them. Then we translate the same task into three programming languages: Python, C++, and C#. In total, we have 45 unique tasks. Although the sensemaking task includes questions that might have also been asked in studies focused on code comprehension [37], the main difference is those studies typically restrict the scope of their questions to either bottom-up [37] or top-down comprehension [38] tasks. Whereas, in our sensemaking task, beside code snippet and question, participants receive also the header of the file with some contextual information, which creates an unusual blend of bottom-up and top-down comprehension tasks which is typically not seen in code comprehension studies which focus on either one or the other. This decision is motivated by our goal of stimulating code exploration, where the participants have to integrate different pieces of information at different locations and create an integrated mental model.

A. Neural Model’s Task

We feed the entire source file of a single task as input, also referred to as prompt, to the generative model and query it for three different answers in the form of text completion. A model processes the input file $p$ by splitting it in tokens via a deterministic tokenizer ($p = t_1, ..., t_n$) and then generates a sequence of tokens as output, as shown on the left of Fig. 2. We allow the models to generate an answer of length 100 tokens at maximum, which is more than enough to respond to all the questions. We use three widely used open source pre-trained models namely: CodeGen [2] in its language-agnostic variant, InCoder [20] and GPT-J [21], all in their largest variants of 16B, 6B and 6B of parameters respectively. To query the model multiple times we use the temperature sampling strategy with a temperature of 0.2.

---

2https://www.geeksforgeeks.org/

3CodeGen-16-multi from https://github.com/salesforce/codegen
### Table I

| Snippet Name | Content | LoC | Question |
|--------------|---------|-----|----------|
| nqueens_Q1   | N queens problem | 78-100 | What does ‘solveNQ(-13)’ return? |
| nqueens_Q2   | 78-101 | What are valid dimensions and values for the array ‘board’? |
| nqueens_Q3   | 78-100 | How would you expect the run time of ‘solveNQ(n)’ to scale with ‘n’? |
| hanoi_Q1     | Tower of Hanoi problem | 28-49 | How does the algorithm moves disks from the starting rod to the ending rod? |
| hanoi_Q2     | 26-47 | Which is the base case of the algorithm? |
| hanoi_Q3     | 28-50 | Which is the name of the auxiliary rod in the call TowerOfHanoi(n, ‘Mark’, ‘Mat’, ‘Luke’)? |
| multithread_Q1 | Consumer-producer threads | 106-116 | Is it possible that consumer and producers threads end up in a deadlock state; namely they both wait for each other to finish, but none of them is doing anything? |
| multithread_Q2 | 104-112 | Is there any line of code in the consumer or producer code that will never be executed? If yes, report it below. |
| multithread_Q3 | 104-113 | Will the queue object ever raise an exception in this program? If yes, which condition(s) should be met for the exception to be raised? |
| tree_Q1      | Recursive tree construction | 87-99 | How many calls to ‘constructTreeUtil’ will ‘constructTree([1, 2, 3], [1, 2, 3], 2)’ make? |
| tree_Q2      | 87-99 | Under which conditions could the check ‘if i <= h’ in ‘constructTreeUtil’ be false? |
| tree_Q3      | 89-101 | A part of the code you don’t have direct access to has called ‘constructTree’ with unknown parameters. What can you find out about those parameters? |
| triangle_Q1  | Triangle classification | 66-112 | Which of the functions have side effects (namely it modifies some state variable value outside its local environment)? |
| triangle_Q2  | 66-113 | Which output will you get for the three points [1, 2], [1, 3], and [1, 4]? |
| triangle_Q3  | 66-112 | What could happen if the call to ‘order()’ were omitted from ‘classifyTriangle’? |

---

**IV. Problem Formulation**

The majority of modern large language models (LLMs) are based on the architecture of generative pre-trained transformers (GPT) [39], such as Codex [1], CodeGen [2], and AlphaCode [3]. Self-attention is a mechanism used in these models that allows each processed token to weigh its own importance with respect to other tokens in the same sequence, enabling the model to capture relationships and dependencies within the sequence. In particular, the representation of each token can incorporate information from tokens that come earlier in the sequence, and on the contrary cannot incorporate information from tokens that come later in the sequence. In this work, when

---

**B. Developers’ Task**

We recruit 25 software developers via direct contacts at a large software company, ranging from interns to more senior software engineers, thus having diverse degrees of familiarity with software development and programming. We track the eye gaze of each participant during a 19 minutes session (on average) while they answer as many questions as possible, typically three or four. We ensure they see each main code snippet only once to avoid bias in answering a question on a snippet they have already explored in a previous task. The eye-tracking setup is calibrated at the beginning of each task to ensure consistent data collection.
a token $A$ incorporates information from another token $B$, we say that $A$ attends to $B$, or equally that token $A$ pays attention to token $B$. This attention is usually quantified by a scalar value, called attention weight, which is computed by the model in its attention mechanism.

When the model takes as input a sequence of $x$ tokens, the attention mechanism is applied to each token in the sequence. Fig. 2 on the left shows a toy example with a model of three layers and two attention heads, together with the attention generated by the model. For each token, the attention is computed sequentially through the Layers of the neural model and, at each layer, the attention is computed in parallel $H$ times, once for each sub-network called attention head. Fixing a combination of layer and head, the attention given by $i$-th token to the other tokens of the sequence can be represented by a vector of weights: $a_i = (a_{i,1}, a_{i,2}, ..., a_{i,i}, 0, i_{i+1}, ..., i_{x})$ where $a_{i,j}$ is the weight given by token at position $i$ to token at position $j$. Note that the token cannot attend any token that come later in the sequence, thus the weights $a_{i,j}$ are zero for $j > i$. Stacking the attention vectors one after the other as row, we obtain an attention matrix $A = (a_1, a_2, ..., a_x)$ for the specific combination of layer and attention head, note that it is a lower triangular matrix.

Thus, when the input file comprising $n$ tokens ($t_1, ..., t_n$) is fed to the model $f$, beside a predicted answer of $m$ newly generated tokens ($t_{n+1}, ..., t_{n+m}$), the model also computes an attention tensor $A$ of shape $(L, H, n + m, n + m)$, where $L$ is the number of layers and $H$ is the number of attention heads. In particular, when comparing developers’ and the model’s attention, we focus on studying the attention weights referring to the prompt tokens only, even if some post-processing approach may use the entire tensor $A$.

Note that by construction, not all tokens can attend to all other tokens, thus we define the notions of followers of a token $t_i$ as the set of tokens that can pay attention to $t_i$. This set is defined as $F(t_i) = \{ t_j \mid j > i \}$, where the subscript represents the position of the token in the sequence.

**A. Views of Attention**

In our problem formulation, we model an extraction function $g$ that takes as input the attention tensor $A$ and returns either a measure of how much attention the model pays to each part of the prompt or a measure of how much each part is linked to other parts of the prompt. Depending on the case, we refer to the outputs as visual attention vector or interaction matrix respectively.

1) **Visual Attention Vector:** It is a static view telling us which part of the input is important for the model when solving the sensemaking task. We define a visual attention of a model as a vector $a = (a_1, ..., a_c)$ over the $c$ characters of the prompt, where each $a_i$ intuitively tells us how much attention was given to that the $i$-th character when solving the task. We use $g_{\text{viz}}(A)$ to model a function that takes as input the attention tensor $A$ and returns a visual attention vector $a$.

2) **Interaction Matrix:** It is a dynamic view that tells us, given a position in the prompt, which other position of the prompt is more deeply connected to it. We define an interaction matrix $S$ as a right stochastic matrix with size $n \times p$ where $n$ is the number of tokens in the prompt and $p$ is the number of admissible target positions in the prompt. We distinguish two kinds of interaction matrices depending on the granularity of the target position $p$, either pointing to another token or line in the source code (the latter being of interest primarily with developer tooling in mind, which is often line based). Respectively, we call them: (1) token-level, where $S$ has size $n \times n$ where $n$ is the number of tokens in the prompt; (2) line-level, where $S$ has size $n \times n_L$ where $n_L$ is the number of lines in the prompt. We use $g_{\text{token}}(A)$ and $g_{\text{line}}(A)$ to model two functions that take as input the tensor $A$ and output an interaction matrix, either $S_{\text{token}}$ or $S_{\text{line}}$ respectively.

**V. Extraction Functions**

We investigate two algorithms for extracting the visual attention vector and four for the interaction matrix.

**A. Attention Extraction Overview**

Fig. 2 illustrates the process of querying the model and extracting its attention signals, leading to the comparison with human developers. In particular, it shows how to the attention tensor $A$ is derived by querying the model, and how to post-process it to extract both the interaction matrix $S_{\text{token}}$ and the visual attention vector $a$. It is split in three phases: generation, post-processing, and human comparison. In the generation phase, a neural model with $L$ layers and $H$ attention heads processes $n$ prompt tokens to generate both $m$ tokens and an attention tensor $A$. Here, the model has three layers and two attention heads, handling five prompt tokens and generating four new tokens. During the post-processing phase, the attention tensors are aggregated to form interaction matrices $S_{\text{token}}$ using two techniques: mean aggregation and follow-up attention, as explained in Section V-C. Note that the matrices are cut to consider only the attention to the $n$ tokens of the prompt. Then, the interaction matrix from token-to-token is converted to the line-level interaction matrix by aggregating the attention weights of the tokens belonging to the same line, obtaining $S_{\text{line}}$. At this point, the visual attention vector $a$ is extracted from the $S_{\text{token}}$ matrix (see Section V-B) and converted to the character level. In the last phase, the interaction matrices $S_{\text{line}}$ and the visual attention vectors $a$ are compared with human data collected via eye-tracking.

**B. Visual Attention Vector Extraction**

We introduce two alternative approaches called attention mean and attention max to condense the attention tensor $A$ to the visual attention vector $a$, namely to implement $g_{\text{viz}}(A)$: $A \rightarrow a$. The first approach is visualized in the bottom part of Fig. 2.

1) **Attention Mean:** It aggregates over all the layers $L$ and attention heads $H$ by taking the average attention weight for each token position. After keeping only the prompt tokens, this step outputs a matrix $A$ with shape $(n, n)$ where each element
$A_{i,j}$ is the average attention paid by the $i$-th token to the $j$-th token in across all layers and heads. Note that it is a lower triangular matrix because a token cannot attend tokens coming after it by construction. Then, we compute the mean of each column excluding the zeros to avoid penalizing more recent tokens with fewer followers. This step corresponds to represent each token $t_i$ with the average attention given to it by its followers $F(t_i)$, thus we call the step mean of followers. It outputs a token-level visual attention $\mathbf{a}$ vector that is converted to character-level vector, by dividing the attention weight on a single token in equal shares among all its characters.

2) Attention Max: This approach differs from the previous one in how it condenses layers and heads in the first step, replacing the mean with the max function to favor the extremely positive signals appearing only in one or few layers and heads; the rest is unchanged.

C. Interaction Matrix Extraction

We study four approaches: mean, max, rollout and follow-up attention. Apart from the rollout attention, which has been introduced by [40], the other three are either inspired by the work of [8] or a novel contribution of this work, such as the follow-up attention.

1) Attention Mean: It computes the mean among all the $L$ layers and $H$ attention heads: $g_{mean} = \frac{1}{LH} \sum_{l=1}^{L} \sum_{h=1}^{H} A_{i,h}$.

2) Attention Max: It computes the max among all the $L$ layers and $H$ attention heads: $g_{max} = \max_{l=1}^{L} \max_{h=1}^{H} A_{i,h}$.

3) Rollout Attention: It propagates the information contained in the attention weights layer by layer from input to output by multiplying the attention weights along multiple paths starting and ending in the same input-output pair. Since it does not model the attention head dimension, we condense that dimension via a simple sum. There is no a priori criterion for which attention layer should be used in the end. Thus, we sum the rollout values of all the layers [40] for more details.

4) Follow-Up Attention: It is our novel approach that centers on modeling the flow of information between subsequent layers. The intuition behind it is that follow-up attention tracks whether a token being attended to in one layer will cause a different token being attended to in the next layer. This is the model analogue of tracking humans jumping from attending one token to attending a different token next.

Algorithm 1 summarizes the entire procedure, which is also represented in Fig. 2. Similarly to the rollout attention, we aggregate the attention weights over the $H$ attention heads by summing the weights of $\mathbf{A}$ along the attention head dimension and obtaining a layer-wise attention $\mathbf{L}$, a 3-dimensional tensor (Line 3). The follow-up attention explicitly models the temporal relationship between the attention weights computed at different layers since the attention weights in the later layers depend on the earlier ones. The intuition is that the layer-after-layer transformation reflects how the models explore code through time, similar to multiple successive fixations of a developer when navigating and exploring source code. Instead of looking at how the token gives attention to other tokens in the same layer, the follow-up attention adopts a differential approach which compares the attention received by token $i$ at layer $z$ with the attention received by token $j$ at layer $z - 1$. To represent this received attention, we define the follower score $f_i^z$ of token $i$ at layer $z$, as the vector of the attention quota that each other token (which we call observers) gives to token $i$ at the same layer (Line 9). Note that, similarly to the attention vector, the follower score is also a vector of real numbers and it has the same length corresponding to the input sequence length, thus representing a complementary viewpoint. To realize the agreement between follower scores at two consecutive layers, we use the cosine similarity as a soft version of the intersection of the set of followers of the two tokens (Line 11). Then we compute the follow-up attention for each ordered pair of tokens $i \rightarrow j$ (Lines 6–7) and for each pair of consecutive layers (Line 4) and condense all layer pairs into a single matrix via sum (Line 13). We aggregate attention over multiple layers since [41] have empirically shown how token identifiability is retained over layers, thus a generic embedding at position $e_l$ in any layer $l$ is traceable to the input embedding $x_i$ in the input sequence.

VI. CODE EXPLORATION DATASET

We conduct an eye-tracking study and collect a novel dataset comprising 25 participants across 92 valid sessions. Participants’ professional software development experience is summarized in Table II. Additionally, 24% of participants have more than 4 years of experience. The dataset contains 17.4% (16) of
Each session is recorded in Visual Studio Code 5 to have a millisecond. The average of fixations per session is 603.66.

To map the pixel of each eye fixation event to a specific character code area during the experiment. Then, for each timeframe, we ensure that we have a consistent grid mapping between pixel positions and character positions in the code area by using a grid over the character positions identified by a line and column.

To derive the developer attention maps from the eye tracking data, we first synchronize data from the VSCode plugin and the eye tracker, via their timestamps. Then, we convert the fixation point of gaze \(x\) and \(y\) coordinates of each \(\text{evt}_{\text{eye}}\) to the corresponding character line and column coordinates in the relative coordinate system of the code area. And knowing the line number \(l\), we can attribute the fixation to a specific character position in the original file. In this way, we convert each \(\text{evt}_{\text{eye}}\) to its equivalent event in character coordinates \(\text{evt}_{\text{char}} = (t, c, l, d)\) where \(t\) is the timestamp, \(c\) and \(l\) are the column and line coordinates with respect to the original file of the fixation point and \(d\) is the duration of the fixation.

Since it is hard to tell whether, during a single fixation event, a participant is looking at a specific character or a group of neighboring characters, we attribute the developer’s attention to neighboring characters. In particular, if the developer looks at position \((c, l)\) in the original file, we augment our data by introducing new events which point to all the neighboring characters within a vertical offset \(v_{\text{off}}\) and a horizontal offset \(h_{\text{off}}\) from our coordinate \((c, l)\). As a result, we replace each basic \(\text{evt}_{\text{eye}}\) with the set of derived events \(\text{evt}_{\text{char}} = (t, c_{\text{new}}, l_{\text{new}}, d)\) where \(c - h_{\text{off}} \leq c_{\text{new}} \leq c + h_{\text{off}}\) and \(l - v_{\text{off}} \leq l_{\text{new}} \leq l + v_{\text{off}}\). This is strictly connected to the concept of fovea region. Indeed, as reported by [42], our fovea region, which is responsible for a sharp central vision, accounts for 2° of the visual field, whereas the parafoveal region, which is used for visual search and scene perception, accounts for 5° of the visual field. Thus considering 5° visual region and our screen size (527mm x 296mm), the developer can see 7.16 characters horizontally and 2.92 characters vertically. Rounding those quantities we set \(v_{\text{off}} = 1\) and \(h_{\text{off}} = 4\). This approach also contributes to mitigating any small \(x\) and \(y\) errors in the eye-tracking data collection.

B. Ground Truth Visual Attention

Here we borrow from the concept of human attention proposed by [8] and define the analogous developer attention as the total time that a specific char was visible to the participant (i.e., in their field of vision): \(d = (d_1, ..., d_e)\) where \(e\) is the number of characters in the prompt and \(d_i\) is the total time that the \(i\)-th character was visible to the participant according to the eye tracking data. In contrast to [8], we consider the char-level instead of the token level because it is more natural for our eye-tracking data.

C. Ground Truth Interaction Matrix

From each developer session, we derive a ground truth interaction matrix \(S\). For a fair comparison of neural models with developers, we take into account the tokenization used by the neural model, namely we use the CodeGen tokenizer\(^7\) which is based on byte-level byte-pair-encoding [44].
To convert char-level events into token-level ones, for each timestamp, if at least one character of a given token is visible, then the token is considered visible as well and we count the corresponding event $e_{\text{eye}}^{(\text{token})} = (t, i, d)$ where $t$ is the timestamp, $i$ is the token index and $d$ is the event duration. Based on the pairs of events involving token $i$ and token $j$, we quantify how likely it is that the developer looks at token $j$ after having looked at token $i$.

Intuitively, we want to have stronger connection when a fixation on token $i$ is shortly followed by a fixation on token $j$, and if this second fixation has a significant duration. Thus, we define the strength of the temporal connection between token $i$ and token $j$ as:

$$S_{i,j} = \sum_{e_{\text{evt}} \in P_{i,j}} \int_{t_i}^{t_j+d_j} e^{-\alpha(t-(t_i+d_i))} \, dt$$  \tag{1}

where $P_{i,j}$ is the set with all the pairs of events where token $i$ is seen before token $j$ and the discounting factor $\alpha$ controls the decay of the connection the more the two events are apart in time. For our experiments we empirically set $\alpha = 0.1$, accounting for observed behavior where developers often spend several seconds scrolling and presumably shallowly searching the code. In Fig. 3 we show an example of the integral connecting two consecutive events.

We noticed a strong neighboring effect across the whole dataset, where the connection between closer tokens tends to be relatively stronger irrespective of context and content. Indeed, developers do not jump randomly between likely locations within the code: they have a significant bias for staying close to their current position. We, therefore, begin by attempting to predict the observed strength of the temporal connection between tokens solely on the basis of their relative position. We suggest a two-tiered approach: first, consider whether the developer is traversing forwards or backward, then use a relative model for how far they will move in that direction. We expect the ratio of forwards or backward traversal to be dependent on the exact task, and in fact, in our dataset the proportion of forward traversal ranged from 45.8% to 78.2%. For each task individually, as well as to some extent in general, the best simple predicting feature for traversal direction appears to be the current token position divided by the total number of tokens, i.e. the ratio of the document still in front of the developer. Fitting individual linear regressions for going forward (Eq. 2) and backward (Eq. 3) (which do not sum up to 1 because of the chance of returning to the token itself) gives the predictions of

$$\sum_{j<i} S_{i,j} \approx 0.94064 - 0.74599 \ast i/\max(i) \quad (R^2 = 0.56) \quad (2)$$

$$\sum_{j>i} S_{i,j} \approx 0.05261 + 0.74575 \ast i/\max(i) \quad (R^2 = 0.56) \quad (3)$$

where the $R^2$ values indicate the goodness of fit, $i$ is the token index, $\max(i)$ is the total number of tokens in the document, and the numeric values are the coefficients of the linear regression. We expect, and find, the distribution for the distance between consecutive gaze points to be less dependent on the task. Of a number of standard distributions we tested against (normal, poisson, lognormal, exponential, Pareto, Weibull), it is best modeled using a fitted Weibull distribution, with the best fit of shape = 0.89, scale = 98.14 tokens going forward and shape = 0.88, scale = 105.61 tokens going backward (see Fig. 4).

Thus, for the code exploration task, to extract relevance beyond mere closeness, we normalize each row of the interaction matrix $S$ by dividing by the average empirical ground truth distribution where the probability to go to a token constantly decreases the further away the target token is.

VII. RESULTS

In this section, we compare the visual attention and interaction matrix extracted from the attention tensor of neural models against the ground truth computed from the developers. We organize our empirical investigation in the following research questions:

- **RQ1**: How effective are developers and neural models in solving sensemaking tasks?
- **RQ2**: How does the visual attention of developers and neural models compare?
- **RQ3**: How is the agreement between developers and neural models influenced by the programming language?
- **RQ4**: How do the interaction matrices of developers and neural models compare?
- **RQ5**: How is the effectiveness of follow-up attention influenced by layer choice and number of newly generated tokens?
RQ1 and RQ2 considers three neural models: CodeGen\(^8\), GPT-J\(^9\) and InCoder [20]. Whereas, for the remaining questions we focus on the larger and more effective CodeGen model.

A. RQ1: Answer Correctness

To evaluate the effectiveness of the developers and models in solving the sensemaking task, we annotate each generated answer by both groups involving four annotators in the process. We use a scale of three values of correctness: (1) correct, when the answer touches all the expected correct points, (2) partial, if at least part of the correct answer is present or if the answer is wrong but in the same style of the correct solution (e.g. the Big-O notation), (3) wrong, when the answer does not contain any correct part. Note that, especially for the model, if the model generates extra text beyond the correct or partial answer we ignore the rest if it is incorrect. Moreover, whenever the question is under-specified we accept multiple correct answers as long as they are compatible with the question. To ensure a reproducible annotation process, all the authors collectively come up with a shared set of gold-standard answers for each question. Then, two of the authors independently annotate more than 20% of the answers generated by the model and the developer, and within two rounds of annotation followed by discussion, the final set of gold standard answers is agreed upon. The final agreement on the 20% of data led to a Cohen’s Kappa of 0.711, 0.898, 0.833 and 0.783 for developers, CodeGen, Gpt-J, and InCoder respectively, which is considered a very high agreement [45]. Finally, the remaining 80% of the data is split in half and annotated only by one of the two authors individually. The upper part of Fig. 5 shows the percentage of correct, partially correct, and wrong answers for each developer and neural model.

Answer to RQ1: The developer’s answers are correct or partially correct in almost 70% of cases, while the neural models under study, namely CodeGen, Gpt-j, and InCoder, show a promising and non-trivial performance (respectively 39.3%, 36.3%, 37.7% correct or partially correct answers).

B. RQ2: Agreement on Visual Attention

To measure the agreement between the visual attention of developers and the neural model, we regard them as vectors with meaningful ordinal content and compute their *Spearman rank correlation coefficient* [46], aligned with related work [8]. In Fig. 6, we report the Spearman rank correlation between the developer attention vector and the model attention vector. For completeness, we also report the comparisons among developers. Note that we only compare the attention maps of two different subjects from the different groups (e.g. developer vs CodeGen, developer vs GPT-J, etc.) when looking at the same code snippet and question. Moreover, neither the data from the participants, nor those extracted by multiple model predictions are aggregates among subjects of the same group. Instead, we consider the comparisons of all the possible combinations of subjects from the two groups. We avoid aggregation because it may be sensitive to largely deviating data of single participants and the identification of a suitable aggregation function is a non-trivial task, as reported by [47].

Related work [8] avoids aggregation for similar reasons. This approach is adopted in all the comparisons among subjects in the paper. The observed agreement exceeds that observed in previous work [8], which we hypothesize to be due to a combination of us using a more advanced model [48] and a more natural data collection setup (eye tracking vs a deblurring interface). To investigate whether higher model-human agreement is connected to higher effectiveness on the sensemaking task we compare agreement of the cases where both human and model are correct and where they are both wrong. We run a statistical t-test and similarly to [8] we find that for InCoder the comparisons where both human and model are correct have a higher agreement than those where they both are incorrect (pval=2.23e-11). We use the t-test under the assumption that the agreement values are normally distributed, which we confirmed

---

\(^8\)https://huggingface.co/Salesforce/codegen-16B-multi

\(^9\)https://huggingface.co/EleutherAI/gpt-j-6B
TABLE III

| Model | Lang-A | Lang-B | Stat. pval | Result |
|-------|--------|--------|------------|--------|
| Codegen | cpp | cs | 1486.0 | 9.95e-07 | Diff. |
| Gptj | cpp | cs | 3404.0 | 6.61e-02 | Same |
| Incoder | cpp | cs | 1280.0 | 1.96e-08 | Diff. |
| Codegen | cpp | py | 2685.0 | 7.22e-01 | Same |
| Gptj | cpp | py | 2571.0 | 9.37e-01 | Same |
| Incoder | cpp | py | 2947.0 | 1.73e-01 | Same |
| Codegen | cs | py | 10334.0 | 9.29e-15 | Diff. |
| Gptj | cs | py | 5323.0 | 2.00e-02 | Diff. |
| Incoder | cs | py | 10327.0 | 1.04e-14 | Diff. |

by visual inspection. For the other two models, there is no statistical significance.

**Answer to RQ2:** The attention of neural models trained on code like CodeGen and InCoder exhibit a significantly higher agreement (+0.22, +0.20) with the developers when answering sensemaking questions as compared to GPT-j (+0.04), which was mainly trained on natural languages text.

**C. RQ3: Programming Language Analysis**

We investigate the differences in the agreement between developers and models across programming languages. In the lower part of Fig. 5 we report the answer correctness (RQ1) divided into groups across the three programming languages under study: C#, C++, and Python. Since each developer has participated in the study only using a single programming language, the difference in answer correctness of developers might be a result of both the programming language or the skill and background of the specific developer. On the other side, the neural models are equally applied to the different languages, and thus we expect that the difference in answer correctness is due to the specific programming language.

We compare the agreement between developers and models across programming languages with the Mann-Whitney U test [49] to compare two distributions. Table III illustrates that the agreement between developers and models is significantly higher on C# than on Python (p-value < 0.05), and marginally significantly higher on C# than on C++ (p-value < 0.1), making C# the language with the highest agreement between developers and models.

**Answer to RQ3:** The effectiveness in answering sensemaking questions is influenced by the specific programming language in which the task is formulated to the neural model, with a gap of up to 13.3 absolute points. Whereas, the agreement between developers’ attention and neural models is higher for C# than for Python, and marginally higher for C# than for C++.

**D. RQ4: Agreement on Interaction Matrix**

Considering CodeGen’s better answer effectiveness and higher developer agreement, we restrict our subsequent investigation to this model.

To compute the agreement between the interaction matrices we use a row-by-row approach, thus considering the task from the perspective of the starting location and asking ourselves: *to which code location should I look next, given that now I am looking at this token?* As a possible target code location, we consider the other lines in the code, thus we compare line-level interaction matrices $S_{line}$. We focus on lines for two reasons: a) because it is well-known that LLMs attend to different kinds of tokens within a line than humans, such as punctuation or newlines [8], and b) because we argue that the line-level is perhaps the most useful granularity from a hypothetical user perspective (see Section X). To obtain $S_{line}$, we take $S$ and sum the probabilities referring to tokens on the same line, thus the probability to go to a line $x$ is the probability to go to any token of that line. To compare corresponding rows of the ground truth interaction matrix and that derived by our attention signal, we use both the Spearman rank correlation coefficient or the top-3 overlap, defined as the number of top-3 target positions shared between the ground truth row and the model-derived row. Moreover, to balance the fact that some potential starting tokens might be rarely (or only very transiently) looked at by the developers, we weight each comparison based on the total number of seconds spent by the developer on the corresponding starting token. We also fix a maximum for this weight to 10 sec to prevent long-observed tokens from dominating the comparison.

We run the different extraction functions on the model attention signal to obtain interaction matrices $S_{line}$, which we compare to developer-derived ground truth. We distinguish between: (1) *attention-based code traversal* predictions, which are those introduced in Section V-C, and (2) *attention-agnostic code traversal* predictions. The attention-based methods comprise raw attention in the first and last layer, max, and mean, with their respective symmetric versions where the triangular matrix is mirrored and added to replace the zero values, the rollout, and follow-up attention. The attention-agnostic methods comprise: *copycat* recommending all the positions containing tokens identical to the starting token (e.g., starting from token `print` it recommends all other lines containing `print` with equal weight), *uniform* recommending all the positions preceding the current token, and *position* recommending the neighboring positions of the current token with a Gaussian distribution centered on the current token.

We find that attention-based methods do carry predictive power, and in particular that *follow-up attention performs best* among all methods for both Spearman rank and top-3 overlap (Figs. 7 and 8). We note that a purely position-based approach performs better than the copycat method despite being completely content-agnostic. We attribute this to developers’ tendency to often read source code in (piecewise) linear order as described by [22]. Regarding raw attention, [11] demonstrated deeper semantic information being concentrated in later layers.
Yet for both the triangular and symmetric versions respectively, higher levels appear inferior at predicting eye movement to earlier levels, possibly because such deeper semantic information may not be apparent to developers.

**Answer to RQ4:** The follow-up attention function performs best in predicting the next code location to look at, with a Spearman rank correlation of +0.49 and a top-3 overlap of 47%. This outperforms the baseline prediction accuracy of 42.3%, which uses the session history of another developer to recommend the next line.

### E. RQ5: Ablation Study

We investigate two key design choices for follow-up attention: (1) the selection of layers to use, and (2) the number of generated tokens, i.e., observers. In the top part of Fig. 9, we report the top-3 overlap for different pairs of layers among the 34 available in CodeGen, together with the configuration where all layers are considered on top. In the same lower part of Fig. 9, we show the top-3 overlap when restricting the usage of the next 10, 50, or 100 generated tokens or “followers”. There is significant agreement with the ground truth even for smaller numbers of layers, particularly the very first one. This suggests little processing, and maybe even only the token embedding, might be needed to extract valuable information. Additionally, a higher number of observers of the follow-up has a positive impact on the agreement of the follow-up attention with the ground truth.

**Answer to RQ5:** The follow-up attention benefits more from using the attention signal produced by early layers and performance are robust to the number of generated tokens.

### VIII. THREATS TO VALIDITY

There are potential threats to validity that may limit the generalizability of our findings. First, our sample of developers may not represent all such developers, especially since we recruited from one large technology company. However, we did screen participants and require that they have professional programming experience. Second, the tasks are sensemaking tasks, as opposed to ecologically valid debugging or feature enhancement tasks, involving code that participants are not familiar with, and participants were restricted from running the code, using a debugger, and performing web searches. Such tasks are commonly used in technical interviews and the participants did not indicate the tasks were atypical. Third, reactivity effects may occur since participants knew they were being observed and may believe their technical ability is being assessed. To minimize this threat, we advised participants that their individual performance was not being reported or analyzed and the observations were performed remotely with the researcher not being present in the same physical room as the participant. Fourth, it is possible that a developer’s gaze does not always represent their attention, though such eye-tracking data has been well-studied for decades in numerous domains. Fifth, regarding possible model’s “cheating” due to memorization [50], we acknowledge the impossibility to exclude that these programs have not been seen by the models during training. However, we note that we evaluate them on the performance on the sensemaking task, for which the combination snippet...
and question is novel to the models since it was created for this study. Regarding the attention distribution, whether models exhibit different attention patterns on code snippet that have been during training, as opposed to those that are novel to them, still remains an open question for future work. Finally, the answers generated by the neural models are dependent on the prompts we provide, and thus results may vary with more elaborate prompt design, which is an active research area in prompt engineering [17], [51], [52], [53], [54], [55].

A. Generalizability

In the design of the study, we aimed to make the evaluation as generalizable as reasonably possible and in line with the current state of the art in the field. On the human side, comparing our sample size to what is found in other eye-tracking studies [56], we note that both number of participants (25 participants vs $\mu = 19.6$, $\sigma = 13.5$) and number of programs (15 programs vs $\mu = 7.6$ programs, $\sigma = 17.2$) are in line with other studies. On the contrary, most of the current literature focuses on Java (38%) [56], while we have a more diverse set of programming languages (C#, C++, and Python), possibly making our results more generalizable. On the model side, we picked a diverse set of widely popular models in terms of downloads on HuggingFace: CodeGen (81K+), GPT-J (2.5M+), and InCoder (58K+). Regarding the applicability of the follow-up attention to other models, especially closed-source ones, although current model inference APIs do not expose attention information yet, we note that the vast majority of closed source are transformer-based making the extraction of attention possible in principle.

IX. DISCUSSION AND IMPLICATIONS

Generally, each part of a codebase holds myriad disparate connections to other parts of the codebase in such forms as documentation, calls and tests, pattern and format parallelism, examples, data, and control flow. However, the positive moderate Spearman Rank correlation among developers (+0.56) shows that the developers tend to navigate a single file along similar paths when trying to make sense of it with the same goal, i.e. answering to the same question. To some extent, this points to a common notion representing a relationship of general “relevance” of one location to the other, at least as far as we consider a single file as done in the current study; more work is needed to generalize to larger codebases and across files. At the same time, we also find this general relevance relation in human understanding is reflected in the neural processing of large transformer models of code, which also show a remarkably promising correlation with the human exploration paths (+0.49). Surprisingly, the follow-up attention agrees with the ground truth on what line to look at next even more than the developers agree with each other: 47% vs 42.3% (Fig. 8). Note that the two runner-up are still attention based and they also outperform or match developers’ agreement: 44.6% for Max att. (Sym) and 42.3% for Raw att. (1st) (Sym). This shows that attention-based approaches, and follow-up attention above all, are promising for recommending the next code location to look at. Thus, this motivates further work on the analysis and use of the neural attention layers as promising way to support developers in their code exploration tasks.

A. IDEs With 360° Vision

Developers spend a large part of their time understanding existing code [57], [58], and a central role of advanced code editing environments is to facilitate navigation to relevant places, whether the developers’ involvement is active (e.g., search), passive (e.g., highlighting of identical tokens) or semi-active (e.g., jump-to-definition). Such tools typically rely on a proxy for current developer focus, such as mouse pointer position or the user’s cursor in a code editor. The challenge is to find the locations relevant to that focus location since the possible reasons for relevance are heterogeneous and syntactical methods can only surface a limited number of them. Nevertheless, such tools have been an active research area [59], with results such as Strathcona [60], Suade [61], Team Tracks [62], Navtracks [63], Mylar [64], PFIS [65], Prodet [66], and Hipikat [67]. In fact, Singh et al. evaluated various operationalizations of human attention (i.e., cursor location, which code is visible on screen, and a qualitative human judgment) and its impact on predictive accuracy [68], though they did not include eye tracking data. The high rate of success of follow-up attention of recommending at least one relevant line among the top-3 (47%) shows the effectiveness of the attention of neural in providing one such possible proxy for relevant code locations connected to the current statement. Further research is needed to explore how to best incorporate such a proxy into existing tools, and how to best use it to support developers in their code exploration tasks, e.g. either highlight neural attended lines, offer to link to them, or list them in a side panel.

B. LLMs and Human Collaboration

Although the sensemaking task spans over diverse set of topics and has mostly open-ended questions, CodeGen, the largest LLM studied achieves already non-trivial performance with 39.3% of correct or partially correct answers. This is a promising result for the use of LLMs in supporting developers when reasoning on code, and motivates further research on perhaps more specialized sensemaking questions directly liked to specific traditional software engineering task, such as “Is there a bug in this code?” for bug detection [69], [70] or “Is this code vulnerable to SQL injection?” for vulnerability detection [71].

C. Context Prioritization for LLMs

In existing tooing employing LLMs, such as GitHub Copilot [72], the model can process only limited part of the code at the same time, given by the maximum size of the prompt it can take, also called context window [54], [73]. In practice, heuristics are needed to explore which parts should be included in the prompt. Our findings show how attention-based methods exhibit a moderate positive agreement with human experts, especially on the top-3 next lines (47%), thus could be used to prioritize the context to be included in the prompt. From our ablation study, it is encouraging to see that even with only 10
newly generated tokens, the agreement is still higher than the agreement between developers: 45.2% vs 42.3%. These results suggest that real-life deployment of the follow-up attention as a relevance provider could benefit from two important optimizations to reduce the computational cost: (1) restricting the number of layers to consider to the first two, and (2) restricting the number of generated tokens to consider.

X. CONCLUSION

We presented and shared a novel dataset of eye-tracking data, comprising 92 visual attention sessions of 25 developers when answering sensemaking questions in three popular programming languages (Python, C++, and C#). We confirmed that neural models provide promising but less accurate answers than developers to these questions while paying attention to similar parts of the code. We formalized a new code exploration task of predicting developer code traversal and confirmed the attention signal’s relevance for this task by evaluating multiple processing approaches. Besides evaluating existing approaches on the sensemaking task, we contributed the concept of follow-up attention, which shows the best agreement with the developer on the sensemaking task, we contributed the concept of follow-up attention as agreement between developers: 45.2% vs 42.3%. These results suggest that real-life deployment of the follow-up attention as a relevance provider could benefit from two important optimizations to reduce the computational cost: (1) restricting the number of layers to consider to the first two, and (2) restricting the number of generated tokens to consider.

DATA AVAILABILITY

All our code is publicly available at https://github.com/githubnext/followup-attention and the dataset is available here.

ACKNOWLEDGMENT

The authors thank our colleagues at GitHub and Microsoft for their support and feedback on this work. The authors also thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

[1] M. Chen et al., “Evaluating large language models trained on code,” Jul. 2021, arXiv:2107.03374.
[2] E. Nijkamp et al., “CodeGen: An open large language model for code with multi-turn program synthesis,” in 11th Int. Conf. Learn. Representations, (ICLR), Sep. 2022.
[3] Y. Li et al., “Competition-level code generation with AlphaCode,” Feb. 2022, arXiv:2203.07814.
[4] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in Proc. 3rd Int. Conf. Learn. Representations, (ICLR), San Diego, CA, USA, Y. Bengio and Y. LeCun, Eds., 2015.
[5] A. Vaswani et al., “Attention is all you need,” in Proc. Adv. Neural Inf. Process. Syst., vol. 30, 2017, pp. 5998–6008.
[6] Z. Lin et al., “A structured self-attention sentence embedding,” in Proc. Int. Conf. Learn. Representations, Jul. 2017.
[7] S. Vasishth, S. Upadhyay, G. S. Tomar, and M. Faruqui, “Attention interpretability across NLP tasks,” Sep. 2019, arXiv:1909.11218 [cs].
[8] M. Paltenghi and M. Pradel, “Thinking like a developer? Comparing the attention of humans with neural models of code,” in Proc. 36th IEEE/ACM Int. Conf. Automated Sofw. Eng. (ASE), Nov. 2021, pp. 867–879.
[9] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, “What do they capture? A structural analysis of pre-trained language models for source code,” in Proc. 44th Int. Conf. Sofw. Eng., New York, NY, USA: ACM, May 2022, pp. 2377–2388.
[10] J. Vig and Y. Belinkov, “Analyzing the structure of attention in a transformer language model,” in Proc. ACL Workshop BlackboxNLP: Analyzing Interpreting Neural Networks NLP, Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 63–76.
[11] K. Zhang, G. Li, and Z. Jin, “What does transformer learn about source code?” Jul. 2022.
[12] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in Proc. 6th Int. Conf. Learn. Representations, (ICLR), Vancouver, BC, Canada. Conf. Track Proc., OpenReview.net, 2018.
[13] T. Yuan, X. Li, H. Xiong, H. Cao, and D. Dou, “Explaining information flow inside vision transformers using Markov chain,” in Proc. Explainable AI Approaches Debugging Diagnostics, Oct. 2021.
[14] H. Chefer, S. Gur, and L. Wolf, “Transformer interpretability beyond attention visualization,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit., 2021, pp. 782–791.
[15] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, and K.-R. Müller, “Layer-wise relevance propagation: An overview,” in Explainable AI: Interpreting, Explaining and Visualizing Deep Learning, Lecture Notes in Computer Science, Springer, May 2022, pp. 1–35.
[16] T. Ahmed and P. Devanbu, “Few-shot training LLMs for project-specific code-summilarization,” in Proc. 35th IEEE/ACM Int. Conf. Automated Sofw. Eng., New York, NY, USA: ACM, Jan. 2023, pp. 1–5.
[17] C. S. Xia and L. Zhang, “Less training, more repairing please: Revisiting automated program repair via zero-shot learning,” in Proc. 36th ACM Joint European Sofw. Eng. Conf. Symp. on the Foundations of Soft. Eng., ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 959–971.
[18] D. Fried et al., “InCoder: A generative model for code infilling and synthesis,” in 11th Int. Conf. Learn. Representations, Sep. 2022.
[19] B. Wang and A. Komatsuzaki, “GPT-J-6B: A 6 billion parameter autoregressive language model,” May 2021.
[20] T. Blascheck and B. Sharif, “Visually analyzing eye movements on natural language texts and source code snippets,” in Proc. 11th ACM Symp. Eye Tracking Res. & Appl., ser. ETRA ’19, New York, NY, USA: ACM, Jun. 2019, pp. 1–9.
[21] T. Busjahn et al., “Eye movements in code reading: Relaxing the linear order,” in Proc. IEEE 23rd Int. Conf. Program Comprehension, May 2015, pp. 255–265.
[22] S. Jain and B. C. Wallace, “Attention is not explanation,” May 2019, arXiv:1902.10186.
[23] S. Wiegrefe and Y. Pinter, “Attention is not explanation,” Sep. 2019, arXiv:1908.04626.
[24] M. Tutte and J. Šnaidjer, “Staying true to your word: (How) can attention become explanation?” in Proc. 5th Workshop Representation Learn. for NLP, S. Gella, J. Weibl, M. Rei, F. Petroni, P. Lewis, E. Strubell, M. Sco, & H. Hajishirzi, Eds. [Online]. Association for Computational Linguistics, Jul. 2020, pp. 131–142.
[25] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, “Understanding neural code intelligence through program simplification,” in Proc. 29th ACM Joint Meeting Eur. Sofw. Eng. Conf. Symp. Found. Sofw. Eng., New York, NY, USA: ACM, Aug. 2021, pp. 441–452.
[26] O. Vinyals, M. Fortunato, and N. Jaitly, “Pointer networks,” in Proc. Adv. Neural Inf. Process. Syst., vol. 28, 2015.
[27] R. Turner, M. Falcone, B. Sharif, and A. Lazar, “An eye-tracking study assessing the comprehension of C++ and Python source code,” in Proc. Symp. Eye Tracking Res. Appl., Safety Harbor Florida, New York, NY, USA: ACM, Mar. 2014, pp. 231–234.
[28] T. Beelders, “Eye-tracking analysis of source code reading on a line-by-line basis,” in Proc. IEEE/ACM 10th Int. Workshop Eye Movements Program. (EMIP), May 2022, pp. 1–7.
[29] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, “Eyes on code: A study on Development navigation strategies,” IEEE Trans. Sofw. Eng., vol. 48, no. 5, pp. 1692–1704, May 2022.
[30] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, “iTrace: Eye tracking infrastructure for development environments,” in Proc. ACM Symp. Eye Tracking Res. & Appl., New York, NY, USA: ACM, Jun. 2018, pp. 1–3.
[33] S. Fakhoury et al., “Gazel: Supporting source code edits in eye-tracking Studies,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.: Companion Proc. (ICSE-Companion), May 2021, pp. 69–72.

[34] M. Behroozi, C. Parnin, and T. Barik, “Hiring is broken: What do developers say about technical interviews?” in Proc. IEEE Symp. Vis. Lang. Human-Centric Comput. (VL/HICC), Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–9.

[35] A. J. Ko, R. DeLine, and G. Venolia, “Information needs in collocated software development teams,” in Proc. 29th Int. Conf. Softw. Eng., Los Alamitos, CA, USA: IEEE Comput. Soc. Press, May 2007, pp. 344–353.

[36] T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. - Volume 1, New York, NY, USA: ACM, May 2010, pp. 185–194.

[37] M. Wyrich, J. Bogner, and S. Wagner, “40 years of designing code comprehension experiments: A systematic mapping study,” ACM Comput. Surveys, vol. 56, no. 4, pp. 106:1–106:42, Nov. 2023.

[38] T. D. LaToza and J. W. Veesey, “The relevance of application domain knowledge: The case of computer program comprehension,” Inf. Syst. Res., vol. 6, no. 3, pp. 286–299, 1995.

[39] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” p. 12.

[40] S. Ahmar and Z. Zaidema, “Quantifying attention flow in transformers,” in Proc. Ann. Meet. Assoc. Comput. Linguistics, Stroudsburg, PA, USA: Association for Computational Linguistics, Jul. 2020, pp. 4190–4197.

[41] G. Brunner, Y. Liu, D. Pascual, O. Richter, M. Ciaramita, and R. Wattenhofer, “On identifiability in transformers,” in Proc. Int. Conf. Learn. Representations, 2020.

[42] E. R. Schottler, B. Angele, and K. Rayner, “Parafocal processing in reading,” Attention, Perception, & Psychophys., vol. 74, no. 1, pp. 5–35, Jan. 2012.

[43] W. Barfield and T. A. Furness, “Virtual Environments and Advanced Interface Design,” Oxford, U.K.: Oxford University Press, Jul. 1995.

[44] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in Proc. 54th Ann. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers), Berlin, Germany: Stroudsburg, PA, USA: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725.

[45] M. L. McHugh, “Inter-rater reliability: The kappa statistic,” Biochem Medica, vol. 22, no. 3, pp. 276–282, Oct. 2012.

[46] C. Spearman, “The proof and measurement of association between two things,” Amer. J. Psychol., vol. 100, no. 3, pp. 441–471, 1987.

[47] G. C. Murphy, “Mylar: A degree-of-interest model for software development teams,” in Proc. 29th Int. Conf. Softw. Eng., (ICSE), May 2010, pp. 147:1–147:25.

[48] V. Augustine et al., “A field study on fostering structural navigation with predit,” in Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng., vol. 2, May 2015, pp. 229–238.

[49] D. Cubric and G. C. Murphy, “Hipikat: Recommending pertinent software development artifacts,” in Proc. 25th Int. Conf. Softw. Eng. (ICSE), 2003, pp. 408–418.

[50] A. Singh, A. Z. Henley, S. D. Fleming, and M. V. Luong, “An empirical evaluation of models of programmer navigation,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., 2016, pp. 9–19.

[51] M. Pradel and K. Sen, “DeepBugs: A learning approach to name-based bug detection,” Proc. ACM Program. Languages, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018.

[52] A. Habib and M. Pradel, “Neural bug finding: A study of opportunities and challenges,” Jun. 2019.

[53] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” IEEE Trans. Softw. Eng., vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[54] “GitHub Copilot - Your AI pair programmer,” [Online]. Available: https://github.com/features/copilot

[55] A. Z. Edward Aftandilian, “Crafting code suggestions using large language models,” in Proc. dagstuhl Seminar Program. Lang. Process. [Online]. Available: https://dgs-cookbook/program/d1MnTaUXO3xL4h8hLJlWtY3vwAhQVZVZBBD0x7/edit#slide=id.g1b0c99fa6_0_94

Matteo Paltenghi received the M.Sc. degree in computer science and engineering from the Politecnico di Milano and TU Berlin. He is a Postdoctoral Researcher with the University of Stuttgart with expertise at the intersection of artificial intelligence and software engineering, advised by Prof. Dr. sc. Michael Pradel. Beside his Ph.D., he also worked as a Research Scientist with the GitHub Next and CERN. He is an Active Member of the software engineering community where he serves as a Reviewer (TSE, TOSEM, JSSoftware), and his work was also presented at top conferences like ASE 21, OOPSLA 22, ICSE 23, and ICSE 24. Recently, he was selected as young researchers for participation in the Heidelberg Laureate Forum (HLF 23). For more information, see https://matteopaltenghi.com.

Rahul Pandita received the Ph.D. degree in computer science from North Carolina State University, in 2021. He is a Staff Researcher with the GitHub Inc. where he works on automated Software Engineering with focus on developer tools and developer productivity. He was a Senior Researcher with the Phase Change Software where he developed a Virtual Assistant ‘MIA’ to help COBOL developers comprehend legacy mainframe systems. For more information, see http://rahulpandita.me.
Austin Z. Henley received the Ph.D. in computer science from the University of Memphis, in 2018. He is a Senior Researcher with the Microsoft where he works on the human factors of AI-powered developer tools. He was a Tenure-Track Professor with the University of Tennessee where he led an NSF-funded lab researching developer productivity and taught software engineering courses. For more information, see http://austinhenley.com/.

Albert Ziegler received the Ph.D. degree in mathematics from the Leeds University and has previously worked on developer productivity and diverse industry ML projects. He is a Principal Researcher with the GitHub Next where he works on Artificial Intelligence for the Software Development Lifecycle. He was one of the three original inventors of GitHub Copilot and has since turned to LLM guided tooling in both in the IDE (Copilot NES, Copilot Radar) and the pull request workflow (AI for Pull Requests, Gentest).