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

Matteo Paltenghi, Rahul Pandita, Austin Z. Henley, 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.

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

Matteo Paltenghi is with the University of Stuttgart, Stuttgart, Germany. E-mail: mattepalte@live.it. Work done while at GitHub Next for a research internship. Rahul Pandita and Albert Ziegler are with GitHub Inc, San Francisco, CA, USA. E-mail: {rahulpandita, wunderalbert}@github.com. Austin Z. Henley is with Microsoft Research, Redmond, WA, USA. E-mail: azh321@gmail.com.

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.

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.

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.

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.

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

---

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

```python
# **************************************************
# The following code reasons about triangles in the
    geometrical sense.
class point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def square(x):
    return x * x
def order(a, b, c):
    copy = [a, b, c]
    copy.sort()
    return copy[0], copy[1], copy[2]
...
p1 = point(0, 0)
p2 = point(1, 1)
p3 = point(1, 2)
classifyTriangle(p1, p2, p3)
# Question: What could happen if the call to `order
    ()` were omitted from
# `classifyTriangle`?
# Answer:
```

Fig. 1: Example of sensemaking task with code and question to be answered in the bottom comment. Completely empty lines have been removed for space reasons.

developer has to "make sense" of code to answer the question correctly. One sensemaking task is contained in a single source code file p 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. Figure 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 *Geeks-forGeeks*[2], 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.

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 Figure 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[3], 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.

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.

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

---

[2]https://www.geeksforgeeks.org/

[3]`CodeGen-16-multi` from https://github.com/salesforce/codegen

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'?
hannoi_Q1	Tower of Hanoi problem	28-49	How does the algorithm moves disks from the starting rod to the ending rod?
hannoi_Q2		26-47	Which is the base case of the algorithm?
hannoi_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'?

TABLE I: Code snippets and related questions for each sensemaking task.

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 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. Figure 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 L *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: $\boldsymbol{a}_i = (a_{i,1}, a_{i,2}, ..., a_{i,i}, 0_{i,i+1}, ..., 0_{i,x})$ 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* $\boldsymbol{A} = (\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_x)$ 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 $\boldsymbol{A}$ 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 $\boldsymbol{A}$.

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 $\mathcal{F}(t_i) = \{t_j \mid j > i\}$, 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 $\boldsymbol{A}$ 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.

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 $\boldsymbol{a} = (a_1, ..., a_c)$ 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 $g_{viz}(\boldsymbol{A})$ to model a function that takes as input the attention tensor $\boldsymbol{A}$ and returns a visual attention vector $\boldsymbol{a}$.

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 $\boldsymbol{S}$ 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
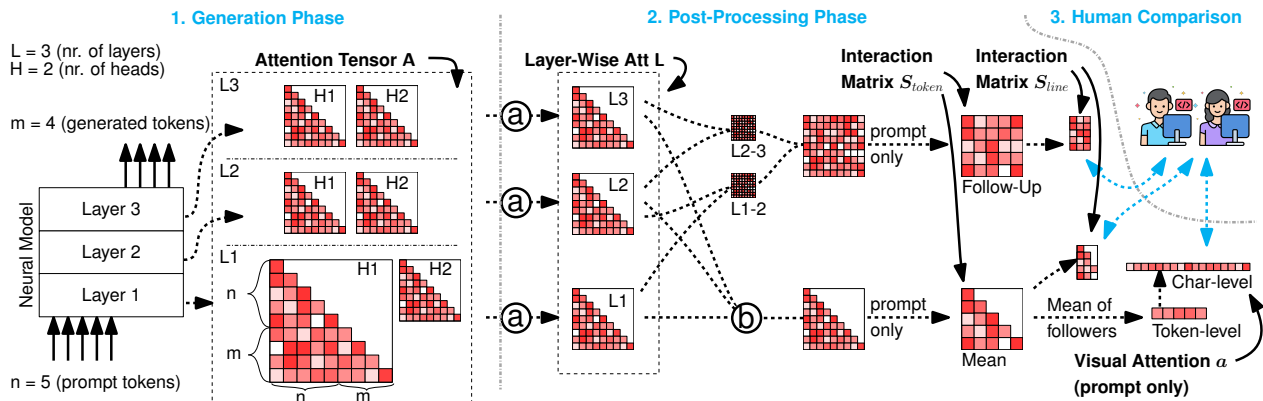
Fig. 2: Overview of the three extraction functions for the visual attention vector and the interaction matrix, both follow-up and mean. Note that a and b represent specific aggregation functions as explained in the text (e.g., mean, max or sum). The darker the red color, the more attention is paid to by token on the row i to the token on the column j.

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 $g_{token}(\mathbf{A})$ and $g_{line}(\mathbf{A})$ to model two functions that take as input the tensor $\mathbf{A}$ and output an interaction matrix, either S_{token} or S_{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

Figure 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 $\mathbf{A}$ is derived by querying the model, and how to post process it to extract both the interaction matrix S_{token} and the visual attention vector $\boldsymbol{a}$. 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 $\mathbf{A}$. 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_{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_{line}. At this point, the visual attention vector $\boldsymbol{a}$ is extracted from the S_{token} matrix (see Section V-B) and converted to the character level. In the last phase, the interaction matrices S_{line} and the visual attention vectors $\boldsymbol{a}$ 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 $\mathbf{A}$ to the visual attention vector $\boldsymbol{a}$, namely to implement $g_{viz}(\mathbf{A}) : \mathbf{A} \rightarrow \boldsymbol{a}$. The first approach is visualized in the bottom part of Figure 2.

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 $\boldsymbol{A}$ with shape (n, n) where each element $\boldsymbol{A}_{i,j}$ 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 $\mathcal{F}(t_i)$, thus we call the step *mean of followers*. It outputs a token-level visual attention $\boldsymbol{a}$ vector that is converted to character-level vector, by dividing the attention weight on a single token in *equal shares* among all its characters.

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.

Attention mean. It computes the mean among all the L layers and H attention heads: $g_{mean} = \frac{1}{l \cdot h} \sum_{l=1}^{L} \sum_{h=1}^{H} \mathbf{A}_{l,h}$.

Attention max. It computes the max among all the L layers and H attention heads: $g_{max} = \max_{l=1}^{L} \max_{h=1}^{H} \mathbf{A}_{l,h}$.

Rollout attention. It propagates the information contained in the attention weights layer by layer from input to output

---

Algorithm 1: Follow-up Attention

1 **Function** FollowUpAttention:
    Input : $\mathbf{A}$; // Dimensions: $(L, H, n+m, n+m)$
    Output: S ;   // Dimensions: (n, n)
2      $CP \leftarrow \emptyset$; // CP: Consecutive layer pairs
3      $\mathbf{L} \leftarrow \sum_{h=1}^{H} \mathbf{A}_h$
4   **for** $z \leftarrow 1$ *to* $L - 1$ **do**
5          $\boldsymbol{S}^{(z)} \leftarrow \emptyset$
6   **for** $i \leftarrow 1$ *to* n **do**
7   **for** $j \leftarrow 1$ *to* n **do**
8                  $s \leftarrow n + m - \max(i, j)$
9                  $\boldsymbol{f}_i^{(z)} \leftarrow \mathbf{L}_{z, s:, i}$
10                  $\boldsymbol{f}_j^{(z+1)} \leftarrow \mathbf{L}_{z+1, s:, j}$
11                  $\boldsymbol{S}_{i,j}^{(z)} \leftarrow \dfrac{\boldsymbol{f}_i^{(z)} \cdot \boldsymbol{f}_j^{(z+1)}}{||\boldsymbol{f}_i^{(z)}|| \cdot ||\boldsymbol{f}_j^{(z+1)}||}$
12          $CP \leftarrow CP \cup \boldsymbol{S}^{(z)}$
13      $S \leftarrow \sum_{\boldsymbol{S}^{(z)} \in CP} \boldsymbol{S}^{(z)}$
14   **return** S

---

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.

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[4], which is also represented in Figure 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* $\boldsymbol{f}_i^{(z)}$ 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 between 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 $\boldsymbol{e}_i$ in any layer l is traceable to the input embedding $\boldsymbol{x}_i$ in the input sequence.

VI. CODE EXPLORATION DATASET

TABLE II: Participants' Professional Software Development Experience

Experience (years)	≤ 1	1-2	2-4	4-6	≥ 10
Number of Participants	8	7	3	3	3

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 sessions on C++ code, 43.5% (40) on C# code, and 39.1% on Python code (36).

The sessions are single-purpose and live-monitored by an experimenter to ensure correct setup and focus on the code exploration task. While each participant has 45 minutes to solve as many tasks as possible, due to calibration and transition times, on average, they spend an average of 18.54 minutes exploring code using the IDE, with an average of 4.92 minutes per single question. A pair of code snippet and question is looked at by a median number of 3 different participants, and each code snippet is looked at by a median number of 7 different participants. No participant is presented with the same code snippet more than once.

Each session consists of a sequence of eye fixation events evt_{eye}, each represented as a tuple (t, x, y, d) where t is the timestamp in milliseconds, x and y are the coordinates of the fixation point in pixels and d is the duration of the fixation in milliseconds. The average of fixations per session is 603.66. Each session is recorded in Visual Studio Code[5] to have a natural coding environment. Based on the size of the parafoveal region [42], each eye fixation event is converted to column and line coordinates: $evt_{eye}^{(char)} = (t, c, l, d)$ where c is the column, and l is the line of the original source file.

A. Eye Tracking Setup

To collect the eye tracking data, we use an eye tracker from GazePoint (model GP3, with $0.5 - 1$ degree of visual angle accuracy), which is placed below the monitor thus not requiring the user to wear any additional device. Note that our setup is as close as possible to a normal coding session without any invasive or unnatural methods. The participants

---

[4]An optimized vectorized version is available here https://anonymous.4open.science/r/attention-study-eye-tracking

[5]https://code.visualstudio.com/

can see between 21 and 26 lines of code. The screen size is 52.7 mm x 29.6 mm with a resolution of 1920x1080 pixels. The participant seats at a fixed distance of approximately 30 cm from the screen. The fixation are computed by the internal fixation filter of the eye tracker, which uses a custom algorithm based on displacement [43], using the `FPOG` (Fixation Point of Gaze) data stream[6] Eye-tracking data are pre-processed using custom code in Python (version: 3.8) described next below and openly shared (see Data Availability).

Besides collecting the eye tracking data evt_{eye}, our setup also collects evt_{txt} coming from a custom VSCode plugin that logs the visible text on the screen. A visible text event evt_{txt} corresponds to a tuple (t, txt, f, l) where t is the timestamp in milliseconds, txt is the visible text, f is the file name shown in the code area, l is the line number of the first visible line with respect to the given file. Note that this event is crucial since we study long code snippets and allow also screen scrolling. To ensure that we have a consistent grid mapping between pixel positions and char positions in the text, we use a monospace font, prevent partial scrolling and prevent any resizing of the code area during the experiment. Then, for each timeframe, we map the pixel of each eye fixation event to a specific character position 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 evt_{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 evt_{eye} to its equivalent event in character coordinates $evt_{eye}^{(char)} = (t, c, l, d)$ 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_{off} and a horizontal offset h_{off} from our coordinate (c, l). As a result, we replace each basic $evt_{eye}^{(char)} = (t, c, l, d)$ with the set of derived events (t, c_{new}, l_{new}, d) where $c - h_{off} \leq c_{new} \leq c + h_{off}$ and $l - v_{off} \leq l_{new} \leq l + v_{off}$. 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 $v_{off} = 1$ and $h_{off} = 4$.
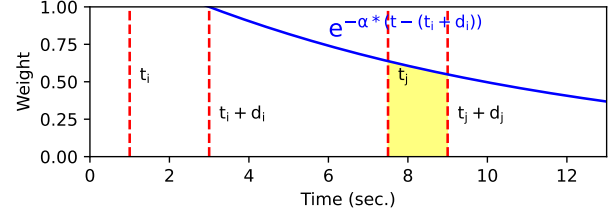

Fig. 3: Example of two events where the yellow area corresponds to their contribution to the connection strength between from token i to token j.

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): $\boldsymbol{d} = (d_1, ..., d_c)$ where c 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 $\boldsymbol{S}$. 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 $evt_{eye}^{(token)} = (t, i, d)$ 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:

$$S_{i,j} = \sum_{evt_i, evt_j \in P_{i \to j}} \int_{t_j}^{t_j + d_j} e^{-\alpha(t - (t_i + d_i))} dt \quad (1)$$

where $P_{i \to j}$ is the set with all the pairs of events where token i is seen before token j and the discounting factor α controls the decay of the connection the more the two events are far apart in time. For our experiments we empirically set $\alpha = 0.1$, accounting for observed behavior where developers often

---

[6]Manual: https://www.gazept.com/dl/Gazepoint_API_v2.0.pdf

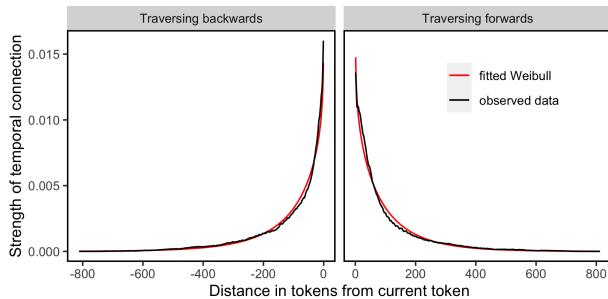[7]https://huggingface.co/docs/transformers/model_doc/codegen

Fig. 4: The strength of the connection $S_{i,j}$ depends significantly on the difference $i - j$. Both cases $i > j$ and $j > i$ can be well modelled using a Weibull distribution.

spend several seconds scrolling and presumably shallowly searching the code. In Figure 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

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

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

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 Figure 4).

Thus, for the code exploration task, to extract relevance be-

yond 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 Figure 5 shows the percentage of correct,
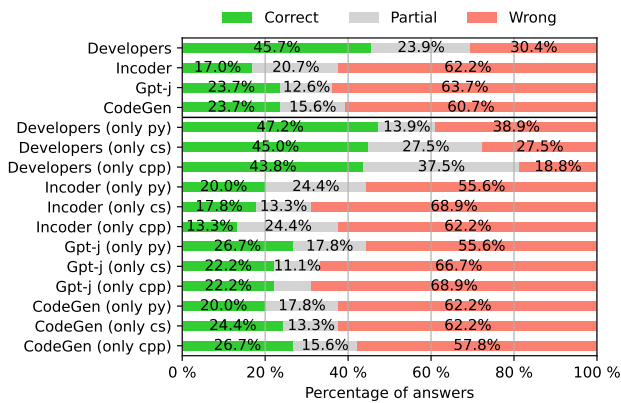
[8]https://huggingface.co/Salesforce/codegen-16B-multi
[9]https://huggingface.co/EleutherAI/gpt-j-6B

Fig. 5: Percentage of correct, wrong, and partially correct answers for developers and model.

Fig. 6: Agreement between developers and models' visual attention (extraction function: max).

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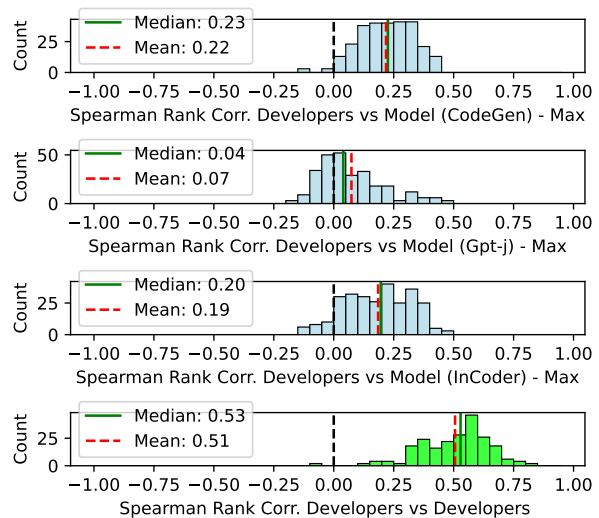
Model	Lang-A	Lang-B	Stat.	pval	Result
Codegen	cpp	cs	1486.0	9.95e-07	Diff.
Gptj	cpp	cs	3404.0	6.61e-02	Same
Incoder	cpp	cs	1280.0	1.96e-08	Diff.
Codegen	cpp	py	2685.0	7.22e-01	Same
Gptj	cpp	py	2571.0	9.37e-01	Same
Incoder	cpp	py	2947.0	1.73e-01	Same
Codegen	cs	py	10334.0	9.29e-15	Diff.
Gptj	cs	py	5323.0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

TABLE III: Results of the Mann-Whitney U statistical tests when comparing the distributions of developer-model agreement (Spearman rank coefficients) across models.

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 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 Figure 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
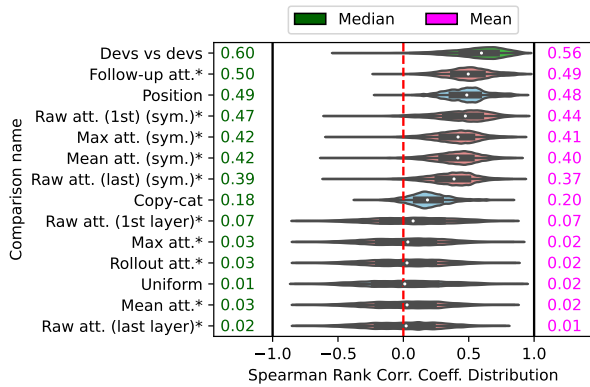
Fig. 7: Spearman Rank Correlation Developers vs Codegen interaction matrix (* for attention-based method) .

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 sense-making 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 (Figures 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 Figure 9, we report the top-3 overlap for different pairs of layers among the 34 available in CodeGen, together with the configuration
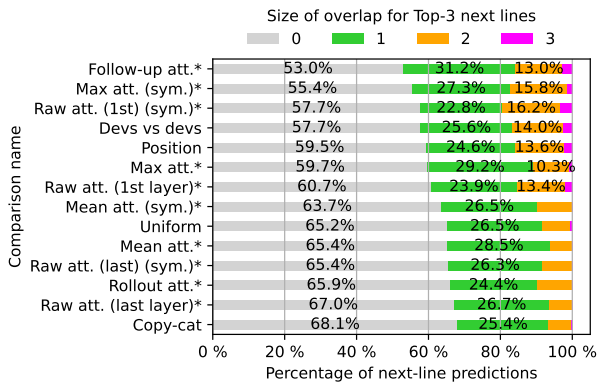
Fig. 8: Top-3 Overlap Developers vs Codegen interaction matrix (* for attention-based method).
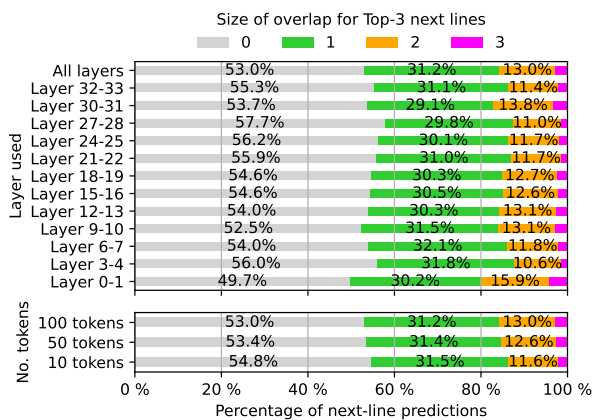

Fig. 9: Effect of layer pair and number of generated tokens on top-3 overlap.

where all layers are considered on top. In the same lower part of Figure 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 the 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 [51, 52, 53, 54, 55, 17].

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% (Figure 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.

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.

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].

Context Prioritization for LLMs. In existing tooling 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* [73, 54]. 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 agreeement 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 attention data.

XI. DATA AVAILABILITY

All our code is publicly available at https://github.com/MattePalte/followup-attention and the dataset is available here [10]

XII. ACKNOWLEDGMENTS

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert,

F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *arXiv:2107.03374 [cs]*, Jul. 2021.

[2] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A Conversational Paradigm for Program Synthesis," Mar. 2022.

[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-Level Code Generation with AlphaCode," *arXiv:2203.07814 [cs]*, Feb. 2022.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All you Need," *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, 2017.

[6] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A Structured Self-Attentive Sentence Embedding," in *International Conference on Learning Representations*, Jul. 2017.

[7] S. Vashishth, S. Upadhyay, G. S. Tomar, and M. Faruqui, "Attention Interpretability Across NLP Tasks," Sep. 2019.

[8] M. Paltenghi and M. Pradel, "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)*, 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 *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 2377–2388.

[10] J. Vig and Y. Belinkov, "Analyzing the Structure of Attention in a Transformer Language Model," in *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for 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 *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 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 *eXplainable AI Approaches for Debugging and Diagnosis.*, Oct. 2021.

[14] H. Chefer, S. Gur, and L. Wolf, "Transformer Interpretability Beyond Attention Visualization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 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*, ser. Lecture Notes in Computer Science, W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, and K.-R. Müller, Eds. Cham: Springer International Publishing, 2019, pp. 193–209.

[16] R. Bednarik, T. Busjahn, A. Gibaldi, A. Ahadi, M. Bielikova, M. Crosby, K. Essig, F. Fagerholm, A. Jbara, R. Lister, P. Orlov, J. Paterson, B. Sharif, T. Sirkiä, J. Stelovsky, J. Tvarozek, H. Vrzakova, and I. van der Linde, "EMIP: The eye movements in programming dataset," *Science of Computer Programming*, vol. 198, p. 102520, Oct. 2020.

[17] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code," Jun. 2022.

[18] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–5.

[19] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," ser. ESEC/FSE 2022, Nov. 2022.

[20] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," Apr. 2022.

[21] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 billion parameter autoregressive language model," May 2021.

[22] T. Blascheck and B. Sharif, "Visually analyzing eye movements on natural language texts and source code snippets," in *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1–9.

[23] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye Movements in Code Reading: Relaxing the Linear Order," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 255–265.

[24] S. Jain and B. C. Wallace, "Attention is not Explanation," *arXiv:1902.10186 [cs]*, May 2019.

[25] S. Wiegreffe and Y. Pinter, "Attention is not not Expla-

nation," *arXiv:1908.04626 [cs]*, Sep. 2019.

[26] M. Tutek and J. Šnajder, "Staying True to Your Word: (How) Can Attention Become Explanation?" May 2020.

[27] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 441–452.

[28] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[29] R. Turner, M. Falcone, B. Sharif, and A. Lazar, "An eye-tracking study assessing the comprehension of c++ and Python source code," in *Proceedings of the Symposium on Eye Tracking Research and Applications*. Safety Harbor Florida: ACM, Mar. 2014, pp. 231–234.

[30] T. Beelders, "Eye-tracking analysis of source code reading on a line-by-line basis," in *2022 IEEE/ACM 10th International Workshop on Eye Movements in Programming (EMIP)*, May 2022, pp. 1–7.

[31] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, "Eyes on Code: A Study on Developers' Code Navigation Strategies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1692–1704, May 2022.

[32] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "iTrace: Eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–3.

[33] S. Fakhoury, D. Roy, H. Pines, T. Cleveland, C. S. Peterson, V. Arnaoudova, B. Sharif, and J. Maletic, "Gazel: Supporting Source Code Edits in Eye-Tracking Studies," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (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 *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2019, pp. 1–9.

[35] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, May 2007, pp. 344–353.

[36] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 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 Computing Surveys*, vol. 56, no. 4, pp. 106:1–106:42, Nov. 2023.

[38] T. M. Shaft and I. Vessey, "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension," *Information Systems Research*, 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. Abnar and W. Zuidema, "Quantifying Attention Flow in Transformers," in *ACL 2020*. Online: 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," *ICLR*, 2020.

[42] E. R. Schotter, B. Angele, and K. Rayner, "Parafoveal processing in reading," *Attention, Perception, & Psychophysics*, vol. 74, no. 1, pp. 5–35, Jan. 2012.

[43] *Virtual Environments and Advanced Interface Design*. Oxford University Press, Jul. 1995.

[44] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725.

[45] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia Medica*, vol. 22, no. 3, pp. 276–282, Oct. 2012.

[46] C. Spearman, "The Proof and Measurement of Association between Two Things," *The American Journal of Psychology*, vol. 100, no. 3/4, pp. 441–471, 1987.

[47] J. Siegmund, N. Peitek, S. Apel, and N. Siegmund, "Mastering Variation in Human Studies: The Role of Aggregation," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 1, pp. 2:1–2:40, Dec. 2021.

[48] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A Transformer-based Approach for Source Code Summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4998–5007.

[49] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar. 1947.

[50] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling Memorization in Code Models," Jan. 2024.

[51] J. A. Prenner and R. Robbes, "Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs," *arXiv:2111.03922 [cs]*, Nov. 2021.

[52] S. Abukhalaf, M. Hamdaqa, and F. Khomh, "On Codex Prompt Engineering for OCL Generation: An Empirical Study," Mar. 2023.

[53] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair," Apr. 2023.

[54] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-Level Prompt Generation for Large Language Models of Code," Oct. 2022.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3445338

15

[55] L. Reynolds and K. McDonell, "Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '21. New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–7.

[56] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "A Survey on the Usage of Eye-Tracking in Computer Programming," *ACM Computing Surveys*, vol. 51, no. 1, pp. 5:1–5:58, Jan. 2018.

[57] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[58] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett, "Foraging and navigations, fundamentally: Developers' predictions of value and cost," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 97–108. [Online]. Available: https://doi.org/10.1145/2950290.2950302

[59] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, July 2010.

[60] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 117–125.

[61] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *Proc. ICSE*, 2007, pp. 780–783.

[62] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 241–248.

[63] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting navigation in software maintenance," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 325–334.

[64] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proc. 4th Int'l Conf. Aspect-oriented Software Development (AOSD)*. ACM, 2005, pp. 159–168.

[65] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart, "Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers," in *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '12. New York, NY, USA: ACM, 2012, pp. 1471–1480.

[66] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Braunlich, and T. Fritz, "A field study on fostering structural navigation with prodet," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 229–238.

[67] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proc. 25th Int'l Conf. on Software Engineering (ICSE '03)*, 2003, pp. 408–418.

[68] A. Singh, A. Z. Henley, S. D. Fleming, and M. V. Luong, "An empirical evaluation of models of programmer navigation," in *IEEE Int'l Conference on Software Maintenance and Evolution*, ser. ICSME '16, 2016, pp. 9–19.

[69] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018.

[70] A. Habib and M. Pradel, "Neural Bug Finding: A Study of Opportunities and Challenges," Jun. 2019.

[71] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[72] "GitHub Copilot · Your AI pair programmer," https://github.com/features/copilot.

[73] A. Z. Edward Aftandilian, "Crafting code suggestions using large language models," 2023, dagstuhl Seminar on Programming Language Processing. [Online]. Available: https://docs.google.com/presentation/d/1MmTaKUOXu7fL6hIjbLvIJ1WIVM3Wy16rQVZ6ZBB0DzY/edit#slide=id.g11be0998ab6_0_94

Matteo Paltenghi is a doctoral researcher at 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 research scientist at GitHub Next and CERN. He is an active member of the software engineering community where he serves as reviewer (TSE, TOSEM, JSSoftware), and his work was also presented at top conferences like ASE 21, OOPSLA 22, ICSE 23 and ICSE 24. Matteo holds a double degree M.Sc. in Computer Science and Engineering from Politecnico di Milano and TU Berlin. Recently, he was selected as young researchers for participation in the Heidelberg Laureate Forum (HLF 23). For more information, visit https://matteopaltenghi.com.

Rahul Pandita is staff researcher at GitHub Inc. where he works on automated software engineering at with focus on devloper tools and developer productivity. He was previously a Senior Researcher at Phase Change Software where he developed a virual assistant 'MIA' to help COBOL developers comprehend legacy mainframe systems. He received his Ph.D. of Computer Science from North Carolina State University in 2015. For more information visit http://rahulpandita.me/
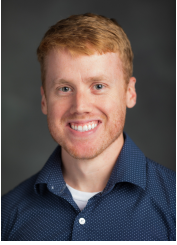
Austin Z. Henley is a Senior Researcher at Microsoft where he works on the human factors of AI-powered developer tools. Previously, he was a tenure-track professor at the University of Tennessee where he led an NSF-funded lab researching developer productivity and taught software engineering courses. He received his Ph.D. in Computer Science from the University of Memphis in 2018. For more information, visit http://austinhenley.com/.

Albert Ziegler is a Principal Researcher at 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).

He holds a PhD in Mathematics from Leeds University and has previously worked on developer productivity and diverse industry ML projects.