Predicting Code Comprehension: A Novel Approach to Align Human Gaze with Code using Deep Neural Networks

TAREK ALAKMEH, University of Zurich, Switzerland
DAVID REICH, University of Potsdam, Germany
LENA JÄGER, University of Zurich, Switzerland
THOMAS FRITZ, University of Zurich, Switzerland

The better the code quality and the less complex the code, the easier it is for software developers to comprehend and evolve it. Yet, how do we best detect quality concerns in the code? Existing measures to assess code quality, such as McCabe's cyclomatic complexity, are decades old and neglect the human aspect. Research has shown that considering how a developer reads and experiences the code can be an indicator of its quality. In our research, we built on these insights and designed, trained, and evaluated the first deep neural network that aligns a developer's eye gaze with the code tokens the developer looks at to predict code comprehension and perceived difficulty. To train and analyze our approach, we performed an experiment in which 27 participants worked on a set of 16 short code comprehension tasks while we collected fine-grained gaze data using an eye tracker. The results of our evaluation show that our deep neural sequence model which integrates both the human gaze and the stimulus code, can predict (a) code comprehension and (b) the perceived code difficulty significantly better than current state-of-the-art reference methods. We also show that aligning human gaze with code leads to better performance than models that rely solely on either code or human gaze. We discuss potential applications and propose future work to build better human-inclusive code evaluation systems.

CCS Concepts: • **Software and its engineering** → *Empirical software validation*; • **Human-centered computing** → *Laboratory experiments*; • **Computing methodologies** → *Neural networks.*

Additional Key Words and Phrases: code comprehension, eye-tracking, neural networks, lab experiment, code-fixation attention

ACM Reference Format:
Tarek Alakmeh, David Reich, Lena Jäger, and Thomas Fritz. 2024. Predicting Code Comprehension: A Novel Approach to Align Human Gaze with Code using Deep Neural Networks. *Proc. ACM Softw. Eng.* 1, FSE, Article 88 (July 2024), 23 pages. https://doi.org/10.1145/3660795

1 INTRODUCTION

Comprehending code is crucial for software developers as it forms the basis for maintaining and evolving code. The easier it is for a developer to understand the code they are working with, the easier it is for them to make the changes and the less likely they introduce any new defects [8, 43, 48]. Yet, how do we know whether a piece of code is easier or more difficult to comprehend for developers and whether the developer is able to understand it correctly? Answers to these questions would allow us to, for example, know how much effort it might take to make a code change, how likely it is to create a bug, or where to best put effort into refactoring.

Authors' addresses: Tarek Alakmeh, University of Zurich, Zurich, Switzerland, tarek.alakmeh@uzh.ch; David Reich, University of Potsdam, Potsdam, Germany, david.reich@uni-potsdam.de; Lena Jäger, University of Zurich, Zurich, Switzerland, jaeger@cl.uzh.ch; Thomas Fritz, University of Zurich, Zurich, Switzerland, fritz@ifi.uzh.ch.

© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART88
https://doi.org/10.1145/3660795

Various measures and approaches have been introduced to address these questions, including metrics such as the cyclomatic complexity [47] or models based on static and other code-related measures to detect quality issues in the code [52, 83]. The majority of these approaches focus on metrics of the code, e.g., its size, nesting, or the change in the code, yet fail to capture the human and in particular, the developer's comprehension process that plays a big role.

One potential way to access developers' cognitive processes when reading code is to measure their eye movements. Eye movements with a high temporal resolution have long been known to reflect cognitive processes in reading natural language [38] and are considered a "window on mind and brain" [78]. Eye movements are categorized into oculomotor events, including fixations (when the eye stays still and obtains visual information, ≈200-300ms) and saccades (fast movements between fixations, ≈30-80ms) [35]. Sequences of fixations, so-called scan paths, have been used as a gold-standard measure in cognitive psychology [59] for many decades.

For code comprehension, researchers have also started to use eye-tracking and study developers' eye movements [4, 37, 68]. Some work has even used eye-related features with machine learning to predict programming proficiency [3]. In our research, we go a step further and aim to explore the potential of using human eye gaze on code to design and train a deep neural network that predicts human code comprehension and perceived code difficulty. Using a deep learning neural network, we explore whether it is feasible to directly extract "hidden" features from a developer's eye fixations and the benefit of aligning it with the code token the developer looked at. Specifically, we address the following research questions:

RQ1 How accurately can we predict (a) code comprehension and (b) perceived difficulty from a reader's eye movements using a deep neural network?
RQ2 How well does our model perform compared to previous models of related work?
RQ3 How much does the alignment of code and fixations contribute to the performance of our model compared to a code-only and fixations-only model?

To address our research questions, we first conducted a lab experiment with 27 participants working on a set of 16 diverse code comprehension tasks while we tracked their eye movements using an eye-tracking system. Since there are generally no standardized code snippet sets [81], we carefully chose diverse code comprehension tasks that are representative of real-world code tasks and fit our experimental constraints. These tasks allowed us to effectively train and analyze our novel deep network architecture using the collected data.

In a second step, we developed and evaluated an end-to-end trained deep neural sequence model using both eye-tracking data and code as input. We exploit the capabilities of natural-language programming-language models (NL-PL) to extract contextualized code-token embeddings and align these code-tokens with gaze data via code-fixation attention. In a comparison with three reference models, our analysis shows that our approach achieves state-of-the-art results, outperforming all three reference models for predicting code comprehension and perceived difficulty. In a short ablation study, we also show that aligning human gaze and code outperforms models that solely rely on either human gaze or code. We discuss potential applications of our approach and future work on extending the approach to build better human-inclusive code evaluation systems that might not even require the active use of an eye-tracker. We make the following contributions:

- The first end-to-end trained deep neural network to predict code comprehension and perceived code difficulty from a developer's eye gaze and the code, together with empirical findings that show that our proposed method outperforms the current state-of-the-art.
- Empirical evidence on the benefit of aligning fixations and code on a fine-grained level for predicting code comprehension.
- An eye-tracking dataset on a carefully selected set of diverse code comprehension tasks.

The rest of the paper is structured as follows: In Section 2, we introduce relevant background information and related work. Our data collection is described in Section 3. Section 4 introduces the machine learning problem setting, which we approach with our novel deep neural sequence model presented in Section 5. We evaluate our deep learning architecture in Section 6. Section 7 discusses our results, while Section 8 analyzes the limitations of our study. Section 9 concludes our work.

All stimuli, anonymized eye-tracking data, and code to reproduce the results are provided online [1].

2 RELATED WORK

In this section, we first start with an overview of code comprehension and studies thereof before discussing research on detecting code quality issues using code-related metrics. We also give a brief overview of measures that capture more of the human in the process before we present an overview of research that uses eye-related data to study code comprehension, also discussing the three reference models that we use as a baseline in our experiments in Section 6.

2.1 Defining and Studying Code Comprehension

A recent systematic mapping study examined 95 studies on code comprehension experiments over the last 40 years showing that there is a lot of ambiguity in the terms used in the domain [81]. For example, terms such as comprehensibility, readability and even legibility are often used interchangeably [6, 14, 79, 80] and need to be defined more clearly as pointed out [23, 81]. In our work, we focus on the comprehensibility of code rather than its legibility that focuses on the non-semantic visual appearance of text or code, or as Oliveira et al. stated it, "the ability of developers to identify the elements of code while reading it" [7, 18, 54].

Code comprehension or its synonym *code understanding* is often referred to as the *process of comprehending code*, especially in literature on the mental models of comprehending code, such as the bottom-up [57, 69], top-down [9, 70] and integrated code comprehension models [46]. At the same time, the term is often used as a *metric to measure how well someone understands the code* [81]. In our work, we refer to code comprehension as the ability to understand code, which can also be referred to as a metric.

To assess whether a developer comprehends a code snippet, prior research has often directly asked a participant to either (a) assess the code themselves by rating it [67], (b) answering comprehension questions about the code [62], (c) summarizing the code [22], (d) locating bugs [21], or (e) evolving the code by either fixing a bug, refactoring or extending it [23, 53, 81]. Since there is no standardized way of conducting code comprehension experiments yet, most studies either take the code snippets from textbooks, open-source repositories, or create them themselves. In our study, we use open-source code snippets, focusing on self-assessment (a) and the answer to comprehension questions (b), since code comprehension itself might not necessarily imply that a developer is able to summarize, refactor, extend, or fix it as previously pointed out [81].

2.2 Detecting Code Quality Issues

Since code quality issues can result in high costs, a lot of research has been trying to automatically assess it. Early on, approaches have focused on static code features to measure the complexity of code, such as cyclomatic complexity or Halstead complexity [6, 44, 47]. While these metrics have their benefits and can easily be applied broadly to determine areas in the code that might be more complex, they fail to capture the human aspect of understanding the code (i.e., comprehensibility). The example in Figure 1 illustrates this shortcoming. It depicts two code examples that both have the same cyclomatic complexity, yet one is generally very easy for a human to understand (right side), while the other is not (left side) [11]. In a study using an fMRI, Peitek et al. [55] also showed

```
int sumOfPrimes(int max) {              // +1
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) { // +1
    for (int j = 2; j < i; ++j) {       // +1
      if (i % j == 0) {                 // +1
        continue OUT;
      }
    }
    total += i;
  }
  return total;
}                  // Cyclomatic Complexity 4
```

```
String getWords(int number) { // +1
    switch (number) {
      case 1:                 // +1
        return "one";
      case 2:                 // +1
        return "a couple";
      case 3:                 // +1
        return "a few";
      default:
        return "lots";
    }
  }      // Cyclomatic Complexity 4
```

Fig. 1. Two code snippets having vastly different comprehensibility, but same cyclomatic complexity [11].

that popular complexity metrics fail to capture comprehension aspects of participants of the code snippets they worked on.

More recent research in the area of automatically detecting quality concerns has also looked into the use of further code metrics [45, 49], metrics on the changes in code [51, 52, 83], or even examined developers' interactions with the code in the IDE [42]. In our research, we complement this research by combining information from developers' eye gaze with code information in a sequence model using a neural network to predict how difficult the code is to comprehend for a developer.

2.3 Using Psycho-Physiological Measures to Capture the Human Element

Several studies have examined the use of various psycho-physiological measures to better capture the human element, including measures on eye-related features such as pupil size, saccades, eye blinks, and fixations gathered from eye-tracking devices, brain-related features such as frequency bands collected with EEG devices, or heart-, skin- and breathing-related measures such as changes in perspiration gathered with EDA and HR sensors. These studies are predominantly feasibility studies and have shown that the captured psycho-physiological measures relate to higher level concepts such as cognitive load [32], memory load [29], mental workload [36, 61], code review performance [68, 76], and also code quality and comprehension difficulty [25, 50]. In our research, we also use psycho-physiological data, in particular eye-related features, and align the data with code tokens in a neural network architecture to predict a developer's code comprehension.

2.4 Eye-Tracking for Code Comprehension

Most related to our research is work that uses eye-related data in the context of code comprehension. Several studies have used eye-tracking devices to investigate aspects such as reading patterns and eye movements in code comprehension experiments [4, 6, 10, 37, 68, 75, 76]. These studies found, for example, that novices look at more code elements than experts [4, 5] and that scanning time inversely correlates with defect detection time [68, 76]. Peitek et al. also went a step further and analyzed area-of-interest-based differences in code reading behaviours using EEG and eye-tracking in combination [56].

Eye-tracking data has also been used in combination with machine learning to predict aspects related to code comprehension. For example, Lee et al. [41] conducted a study with an eye-tracker and an EEG for code comprehension tasks. Using a Support Vector Machine (SVM), they examined the use of features engineered from the collected data to predict task difficulty and a developer's level of expertise. Unfortunately, we could not find sufficient detail or a public implementation for their approach that would allow a comparison.

Our work compares a novel architecture and model with three other state-of-the-art methods in the area. In the method by Fritz et al. [25], the authors examined the use of psycho-physiological features engineered from eye movements, EDA, and EEG data, together with a Naïve Bayes classifier to infer participants' difficulty reading code. In our comparison, we focus on the eye-related features of their method, specifically the fixational, saccadic and pupil features. Different to Fritz et al. [25] that aggregates the features by time, Al Madi et al. [3] used eye-related features aggregated on the word level. By focusing on the word level, the method exploits information about the stimulus but loses the temporal information of the scan path. For their method, Al Madi et al. use eight fixational features (no saccadic information) as input for a Random Forest approach to estimate programming proficiency. Finally, Harada and Nakayama [33] examined the use of several eye-related features to estimate code reading ability using an SVM. In their method, they engineered fixational and saccadic features for larger areas of interest in the code snippets, including transition probability features and power spectra in different frequencies that occur during fixational micromovements (drift and tremor) [19]. To date, no study has explored the ability of deep learning to extract "hidden" features from eye-related data and from the code tokens looked at during the task. We will explore this in the presented study.

3 STUDY METHOD

To create a representative eye-tracking data set on code comprehension and examine our research questions, we conducted an experiment with 27 participants in an eye-tracking laboratory. For the experiment, each participant worked on up to 16 code comprehension tasks while we collected their eye gaze as well as participants' ratings on the perceived difficulty and their response to the tasks. Since the main objective for this experiment was to create a data set that is representative of real-world comprehension tasks and includes a diverse set of code snippets, we applied a multi-step process to identify the code snippets for the comprehension tasks used in the experiment that we will describe in Section 3.4.

3.1 Procedure

The experimental procedure consisted of three phases: an introduction, a main phase, and a debrief. In the *introduction phase*, we introduced each participant to the physical and the virtual environment (i.e., the graphical user interface for the comprehension tasks). The physical environment is depicted in Figure 3 and described below. We then had each participant perform a sample comprehension task to familiarize themselves with the environment, the type of tasks, and the procedure. We also gave them the option to ask any questions they had. Finally, we calibrated the eye-tracker to the participant using a 9-point calibration, requiring a good calibration result with an average offset error of $< 1.0^\circ$ and maximum error of $< 1.5^\circ$ [63, 72].

In the *main phase*, participants were asked to complete a randomly assigned list of comprehension tasks. We predefined 5 lists of comprehension tasks, each with 7 to 10 comprehension tasks, to ensure that (a) each participant works on a diverse and broad range of comprehension tasks including a mix of easy and more difficult ones as well as tasks with and without recursion and iteration, (b) each of our 16 predefined comprehension tasks (see Section 3.4) is performed by several (at least 5) participants, (c) the order of tasks assigned varies between the lists, and (d) the overall time on tasks does not exceed a maximum of 60 minutes per participant. Additionally, since we had several participants performing two sessions, we designed two of the five lists of comprehension tasks to be mutually exclusive from the other three lists. For logistic reasons of the eye-tracking setup, we randomly predefined an order for the tasks within each list. Note that the second sessions were scheduled several weeks after the first one to reduce the burden on participants.

```
class LruCache(object):

    def __init__(self, capacity=3000):
        self.capacity = capacity
        self.cache = collections.OrderedDict()
        self.lock = threading.Lock()
        self.running = True

    def get(self, key):
        with self.lock:
            record = None
            try:
                record = self.cache.pop(key)
                self.cache[key] = record
            except KeyError:
                pass
            return record

    def set(self, key, record):
        with self.lock:
            try:
                self.cache.pop(key)
            except KeyError:
                if len(self.cache) >= self.capacity:
                    self.cache.popitem(last=False)

            self.cache[key] = record

    def __str__(self):
        out_str = ""
        for key, value in list(self.cache.items()):
            if isinstance(value, str):
                out_str += " %s => %s<br>\n" % (key, value)
            elif isinstance(value, dict) or isinstance(value, list):
                out_str += " %s => %s<br>\n" % (key, json.dumps(value))

        return out_str
```

Code Snippet

What will be printed if you run the following code?

```
lru_cache = LruCache()
lru_cache.set("key-3945", "This is a sample value.")
print(lru_cache.get("key-3945"))
print(str(lru_cache))
```

Task Prompt

[A] key-3945
key-3945 => This is a sample value.

[B] This is a sample value.
<LruCache object at 0x11a2734c0>

Answer Options

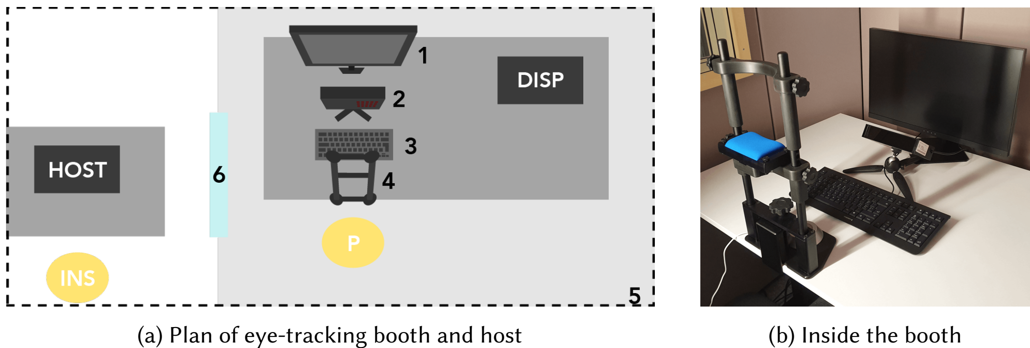
[C] This is a sample value.
key-3945 => This is a sample value.

[D] I don‘t understand the code

Fig. 2. Comprehension task example with the task prompt (top right), the code snippet CS-07 (left), and the answer options (bottom right).

For each comprehension task, participants were presented with a task prompt, the code snippet, and four answer options. The fourth answer option for each task was “I don’t understand the code”. An example is presented in Figure 2. We designed the task prompt of each comprehension task to require the participant’s understanding of the whole code snippet while keeping the prompt short so that the participant has to spend little time reading and understanding it. All task prompts (top right of Figure 2) started with the question “What will be printed if you run the following code?” followed by a very short code snippet (one to a maximum of four lines of code), such as ‘`print(key_value_type(“x=test”))`’, ’`print(pigeon_sort([0,7,4,1,1]))`’, or ‘`n=5; print(foo(n))`’. The main purpose of the task prompt code is to invoke functions of the main code snippet and then print the return value, so that participants spend their time comprehending the code snippet (left side of Figure 2). In the end, participants had to choose an answer option (bottom right). See also Section 3.4 for more details on the comprehension tasks.

We designed the comprehension tasks to be short and solvable in less than five minutes. Keeping the comprehension tasks short enables us to collect a greater number of data points per participant and also accommodates logistical constraints posed by the eye-tracker and the display used for presenting the comprehension tasks. To avoid participants spending too much time on a single



(a) Plan of eye-tracking booth and host

(b) Inside the booth

Fig. 3. Physical setup of the experiment. The participant (P) sat in front of the monitor (1) of the display PC (DISP) with their head stabilized by a chin and forehead rest (4). Their eye movements were recorded by a stationary video-based eye tracker (2). A keyboard (3) was placed on the desk for the participant to interact with the experiment. The participant was seated inside a noise-insulated booth (5) with an observation window (6) for the experiment instructor (INS) who controls the eye tracker via the HOST computer that is synchronized with the display computer DISP.

code snippet, we imposed a 5-minute limit per comprehension task and reminded them to select an answer 20 seconds prior to reaching the time limit. We informed participants that the time limit served a logistical purpose, not to induce pressure or stress, allowing participants to proceed to other tasks without getting stuck on challenging ones. Once an answer was selected or the time limit was reached, participants were presented a short questionnaire asking them (1) to state whether they understood the code snippet (true or false), (2) to state whether they encountered the code or concepts presented in the code snippet beforehand (true or false), and (3) to rate on a 5-point Likert-type [77] scale (1:very difficult to 5:very easy) how difficult the code snippet was for them to understand. The latter will be used as the perceived difficulty score. After completing the questionnaire, participants were shown an aquatic video to help them relax before the next comprehension task was presented, similar to other experiments [25]. We chose videos featuring nature scenery to promote the restoration of emotional and cognitive resources [27].

After participants completed the list of comprehension tasks, we had a *debrief phase*. In this part, we conducted a brief (less than 5 minutes) semi-structured interview with participants. In the interview, we asked participants for general remarks on the experiment and their experience as well as specific questions, such as whether any task was extremely difficult or easy, whether they used the comments in the code to understand the code, and whether they remember looking at blank space to think. The qualitative answers from the brief interviews were merely used to confirm the validity of the study and detect any participant issues (of which there were none). More details on the procedure and the specific comprehension tasks for each of the code snippets can be found online [1].

3.2 Experimental Setup

To collect highly accurate and fine-grained eye gaze data that also reduces any potential distractions, we conducted the experiment in a lab with a booth in which participants performed the comprehension tasks. For eye-tracking, we used the EyeLink Portable Duo from SR Research [60] that provides a sampling rate of up to 2000 Hz, has high accuracy with a spatial resolution offset of 0.01°, records saccades, fixations and blinks, and is compatible with participants wearing glasses or contact lenses. The device works by sending near-infrared light to the eye, causing a reflection on the cornea. The eye's fixation locations are then computed through image processing algorithms using the center of the pupil and the cornea reflection [72].

The overall setup is depicted in Figure 3a with the booth on the right in which the participant (P) sits and the instructor (INS) sitting outside behind a window (6) on the left. Inside the booth, the display computer (DISP) runs the experiment and displays it to the full HD 22-inch 16:9 stimuli screen (1). The eye-tracking device (2) is located in front of the stimuli screen. A keyboard (3) is placed within arm's reach between the eye-tracking device and the head mount (4), which is fixed to a height-adjustable desk. The instructor and participant can communicate with each other via an intercom. Figure 3 depicts a picture of the inside of the booth.

We used the Experiment Builder from SR Research [73] to configure and execute the experiment procedure as described in Section 3.1. This setup allowed us to collect all data from the eye-tracking device as well as the user input via keyboard and additional measures, such as the time to respond, that we configured in the Experiment Builder.

3.3 Participants

We advertised our study among undergraduate and graduate students in computer science or a related field via e-mail and word-of-mouth at two universities. To be eligible, participants had to have at least basic Python programming experience (i.e. having completed an introductory programming course or having equivalent knowledge). Prior to the voluntary participation, all students have been informed verbally and in written form about the study procedure, data collection and privacy, their right to withdraw from the experiment at any time, and their right to data deletion at any time without providing a reason. All participants accepted the conditions by signing a comprehensive consent form. In total, we recruited 30 students and ended up with 27 participants since the calibration showed that the eye tracker was not able to accurately track the gaze of three participants, presumably due to irregular corneas of one or both of their eyes. For the 27 participants, the Python programming experience ranged from 0.5 to 10 years (mean M=2.96, standard deviation ±2.23 years), the professional experience ranged from 0 to 7 years (M=1.80, ±1.82), and the age ranged from 20 to 33 years (M=24.44, ±3.33). 21 participants identified as male, and 6 as female.

3.4 Selecting Comprehension Tasks

To best train and assess the potential and value of the novel deep neural network architecture for predicting perceived code difficulty, and to support its generalizability, we carefully selected the code snippets and comprehension tasks for the lab experiment because there is no standardized way of creating comprehension tasks as mentioned in section 2.1. In particular, we focused on four characteristics in the selection process:

Real-World. Tasks should be representative of real-world tasks developers face in their daily lives.

Diverse. Tasks should cover a diverse set of code snippets, including varying levels of difficulty, and various code constructs used in the snippets.

No Domain Knowledge. Tasks should require no domain knowledge so that we can examine developers' comprehension of the code and reduce the effect of preexisting knowledge.

Fits Physical Constraints. Tasks need to fit within the physical constraints of the experimental setup, especially to ensure a high quality and accuracy of the eye-tracking data.

To create code comprehension tasks that are based on code snippets that exhibit these characteristics, we used a multi-step process detailed in the following. The careful selection process is necessary since the eye-tracking experiment analysis and model training not only rely on the quality of the individual code snippets but also on the quality of the entire set, in particular, its diversity regarding varying levels of difficulty and code constructs used in the snippets.

To select the code snippets, we chose Python as the programming language because it is popular among professional developers and students [20, 30, 64, 74], it is the second-most popular language in Github's trending repositories [30], and also because it is supported by multiple natural language programming language (NL-PL) models [24, 31].

Project and Code File Filtering. To have real-world tasks, we started out by selecting the 500 most popular open-source Github repositories. We then discarded all repositories with less than 30% Python code to only keep projects with a substantial amount of it. From the remaining repositories, we extracted all Python files, ending up with *47'282*. We then filtered out all Python files with less than 10 lines of code (LOC) and more than 100 LOC. From a manual inspection, we noted that files with less than 10 LOC (*5'014* files) often contained only class definitions and variable settings or were empty (e.g. `__init__.py` files), not allowing us to create reasonable code comprehension tasks. Another *22'891* files were larger than 100 LOC, and thus were too long given the physical constraints for the experimental setup and the code having to fit onto the screen. After the filtering we ended up with *20'377* code files.

Filtering for Diversity. To end up with a diverse set of code snippets, we decided to categorize the code files with respect to their usage of recursion and iteration. Since recursion and iteration are fundamental programming techniques, and recursion is considered one of the most difficult concepts to comprehend when learning to program [2, 40], explicitly including these criteria, should result in a diverse set of tasks and varying levels of comprehension difficulty. We, therefore, implemented an abstract syntax tree (AST) walker that we used to check for while/for loop statements and recursive calls. We included files with mutually recursive functions up to the second degree. Using the AST walker, we then classified code files into four categories: (1) *15'138* files (74.3%) with neither while/for loops nor recursive calls, (2) *3'980* files (19.5%) with at least one while or for loop but not recursion, (3) *181* files (0.9%) with both recursion(s) and while/for loop(s), and (4) *105* files (0.5%) containing at least one recursive call but no while/for loop. We excluded 973 files that produced a syntactic error when parsing, caused by erroneous files or external modules. To select a diverse set of snippets, we then randomly selected 50 files from each category, ending up with *200* files. Note that by focusing on the "vanilla" Python language and API, we capture code elements frequently used in most projects yet exclude code involving external library calls or other language features, for which further research is necessary.

Manual Inspection. We manually inspected the remaining 200 code snippets according to our selection criteria. Based on the manual inspection, we then ended up with *18* files, filtering 182 code snippets that required extensive domain knowledge (e.g. relating to frameworks such as django, keras, pytorch or tensorflow), were trivial, or operated on external data. Of the 18 code snippets, 2 were semantically the same (i.e., both performing exponentiation), so we discarded one of the two. In addition, we removed one code snippet to reduce the number of theoretical snippets that are not representative of real-world tasks. We ended up with a final set of *16* code snippets, 5 without loops and/or recursion, 5 with a loop but no recursion, 2 with a loop and recursion, and 4 with recursive calls.

Creating Comprehension Tasks. For each of the 16 code snippets, we then created a comprehension task. We designed the tasks so that they require the participant's understanding of the entire code snippet while also being very short so that the participants have to spend little time reading and understanding the task prompt. In most cases, the tasks are one to at most four lines of code long, calling a function in the code snippet, printing the return value, e.g. `print(foo("x=test "))`, and then asking the participant to select the correct print output from a set of options. An example is presented in Figure 2. For the code snippets with recursion or loops, we designed the tasks

to capture understanding rather than cognitive capacity for going through multiple iterations / recursive steps.

Each comprehension task had four answer options, with one of the four being “I don’t understand the code”. We designed the other three options to not give away the correct answer easily, making all options plausible. We focused specifically on input-output-based single-choice comprehension questions to capture participants’ eye-tracking data for comprehending the entire code file, rather than other tasks, such as bug identification, code extension, or code summarization, that might also include other artifacts in the collected data [81].

In addition to the original 16 comprehension tasks, we created and added up to 2 variations for some of the tasks to add some variation in the way developers write or document code and also slightly vary the burden for comprehending the code. Studies have shown that comments and variable/function names can impact the code’s difficulty and comprehensibility [25]. To encompass a diverse range of difficulty levels, we therefore introduced variations. In particular, we altered the information contained in the names of variables and parameters by obfuscating them and also removed docstrings and comments.

Validation Survey. In a final step for selecting comprehension tasks, we conducted a small-scale survey with four computer science students who did not participate in our eye-tracking experiment. We used the survey to examine whether the selected code snippets capture a diverse range of difficulty levels and asked participants to assess the comprehensibility, feasibility, clarity, and ambiguity of each code snippet and task. Based on the survey responses, we made minor adjustments and were confident that the identified tasks represent a diverse set that varies in the perceived difficulty of comprehending the code snippets, and that the tasks are feasible.

3.5 Data Collection & Analysis

Overall, we recorded 270 data points from the 27 participants and removed 36, ending up with a total of 234 data points for our analysis. One data point represents one participant solving one of the 16 comprehension tasks. We had to remove 36 data points since we initially had experiment sessions in which we used a dark mode display for realism purposes. After we noticed severe inaccuracies in the recorded eye gaze data due to the dark mode, we only used light mode for the rest of the experiment sessions and removed the affected data points. Seven of the 27 participants took part in a second session.

The recorded gaze data amounts to 164’446 fixations and their corresponding features (e.g. position, duration, pupil dilation). After removing fixations outside the code snippet area (i.e. fixations on the task prompt or answer options), we use 129’125 fixations and their features to train and test our model.

The study participants solved $58\% \pm 49\%$ of the tasks correctly, requiring an average of 3.60 ± 2.33 minutes to solve a task. The high standard deviation shows that we have successfully covered a broad range of complexities in our code snippet pool with some tasks being correctly solved in less than 17% of the cases, while other tasks have been solved correctly by everyone (i.e. 100%). Since most tasks were completed well within the time limit, we believe that the time limit had minimal impact on participants.

The participants rated the difficulty of understanding presented code snippets (perceived difficulty) as 2.85 ± 1.13 on average on a 5-point Likert-type scale (1 being very difficult, 5 being very easy). The average comprehension correctness (i.e. correct comprehension task outcome), number of fixations (# fixations), and number of participants who solved the corresponding task per snippet can be found in Table 1. Note that the variation in the number of participants that worked on a given code snippet stems from the assignment of participants to the lists of tasks that we predefined

Table 1. Overview of code snippets and participant's average comprehension performance.

Code Snippet ID	Comprehension Correctness	Perceived Difficulty	Cyclomatic Complexity	Number of Fixations	Lines of Code	Number of Participants
CS - 05 - V2	1.00	4.67	2	124.50 ± 33.53	11	6
CS - 05 - V1	1.00	4.50	2	182.33 ± 63.66	13	6
CS - 01	1.00	4.00	7	224.03 ± 114.26	38	17
CS - 04 - V1	1.00	3.83	5	271.33 ± 145.73	37	6
CS - 09 - V1	1.00	3.45	2	270.91 ± 93.71	26	11
CS - 13	0.83	2.17	3	323.00 ± 95.33	24	6
CS - 02	0.76	2.82	8	534.06 ± 217.31	29	17
CS - 11	0.69	3.38	11	469.77 ± 214.46	37	13
CS - 14	0.62	3.31	14	320.54 ± 260.34	38	13
CS - 07	0.61	2.65	12	496.83 ± 169.05	37	23
CS - 03 - V1	0.60	2.80	4	611.40 ± 154.85	27	5
CS - 08	0.58	2.58	9	566.11 ± 192.68	37	19
CS - 10	0.58	2.08	6	725.00 ± 248.64	23	12
CS - 03 - V2	0.50	2.67	4	594.67 ± 237.38	13	6
CS - 06 - V1	0.50	2.42	3	356.42 ± 156.19	12	12
CS - 09 - V2	0.42	3.00	2	254.00 ± 148.28	2	12
CS - 04 - V2	0.17	2.42	5	759.50 ± 152.22	22	12
CS - 16	0.14	3.14	7	301.86 ± 89.86	16	7
CS - 15	0.14	1.71	5	355.14 ± 84.76	17	7
CS - 12	0.14	1.57	5	594.14 ± 198.58	36	7
CS - 03 - V3	0.08	1.75	4	625.33 ± 250.79	13	12
CS - 06 - V2	0.00	2.00	3	331.40 ± 115.35	12	5

due to logistic reasons (see Section 3.1) and the fact that we were only able to recruit 7 participants for a second experiment session. The table also presents the total cyclomatic complexity calculated using radon [39] and lines of code for each code snippet.

The data also provides evidence that measures such as the cyclomatic complexity do not properly capture perceived code difficulty or comprehension correctness. The Pearson Correlation Test shows that there does not exist a significant correlation between the cyclomatic complexity and comprehension correctness ($r \approx 0.00$; $p > 0.05$). In contrast, the perceived difficulty correlates significantly with the comprehension correctness ($r = 0.76$; $p < 0.05$).

Hereafter, we refer to specific code snippets by their ID that can be traced throughout the entire study and to the code itself found online [1]. A code snippet is identified by its origin ID (e.g. CS-1) and an optional suffix that indicates the readability variation (i.e. V1 for the original, and V2 or V3 for readability variations). When devising the code snippet dataset, we tried to adhere to the action items proposed by Wyrich et al. [81], such as clearly defining our proxy for code comprehension (i.e. solving comprehension tasks correctly), or stating the selection criteria for the code snippets, allowing future research to integrate our dataset.

4 PROBLEM SETTING

We examine two eye-tracking inference problems in code reading: perceived difficulty and comprehension. Given a piece of code and a comprehension question presented together on a screen,

as well as the reader's eye gaze on this stimulus, we want to infer the reader's code comprehension and the perceived code difficulty. Formally, we have a dataset $\mathcal{D} = \{(C_1, E_{1,1}, y_{1,1}), \ldots, (C_c, E_{c,p}, y_{c,p})\}$, where c is the number of code snippets, p is the number of participants, $E_{i,j}$ are the eye movements of reader j recorded during the presentation of the i'th code snippet C_i. The eye movement sequence is represented as a sequence of n fixations, i.e. $E_{i,j} = \langle f_1, \ldots, f_n \rangle$. The predicted label $y_{i,j}$ is either **(a)** reader j's task outcome having value 0 if the task was solved incorrectly and 1 if the task was solved correctly or **(b)** the perceived difficulty of understanding the code snippet C_i perceived by reader j. To model and evaluate both problem settings (i.e., code comprehension and perceived difficulty) uniformly, and since we designed the comprehension task as a binary problem aligned with related work (task solved and therefore comprehended correctly or not), we decided to model the perceived difficulty (b) problem also as a binary classification. For (b) we therefore computed the mean subjective difficulty over all trials and converted the perceived subjective difficulty label to 0 if the reported subjective difficulty was below the mean (i.e. the perceived difficulty was *hard*) or 1 otherwise (*easy*). Thus, given eye movements $E_{i,j}$ of reader j on code snippet C_i we want to predict the reader's (a) comprehension and (b) perceived difficulty of the code snippet $y_{i,j}$.

Since both problems are modeled as binary classification tasks, the model's performance can be characterized by a false-positive rate and a true-positive rate. By changing the decision threshold, one can observe a receiver operator characteristic curve (ROC curve). The area under the ROC curve (AUC) provides an aggregated measure of performance for all possible classification thresholds. It therefore acts as a suitable measure of model discrimination without being tied to a specific threshold (contrary to the f1 score). In summary, while the f1 score is useful for finding a balanced classification threshold, AUC is a more comprehensive measure of a model's ability to discriminate between the classes across all possible thresholds. It provides a global view of the model's performance.

5 A NOVEL DEEP NEURAL SEQUENCE MODEL TO PREDICT CODE COMPREHENSION AND PERCEIVED CODE DIFFICULTY

We developed the first deep neural network architecture that processes a code snippet together with the reader's eye gaze to predict the reader's comprehension or, alternatively, the perceived difficulty of the given code. Our proposed architecture consists of several building blocks: a natural-language programming-language (NL-PL) model, a neural attention mechanism that aligns the code stimulus with the fixation sequence (Code-Fixation Attention), a neural sequence model that processes the fixation sequence together with the neural representation of the stimulus code snippet, and, to compute the final prediction, a fully connected neural network with sigmoid output. An overview of the architecture is presented in Figure 4.

Natural-Language Programming-Language Model. NL-PL models such as CodeBERT can process both natural-language and programming-language. They are commonly utilized for tasks such as code summarization or documentation. Pre-training of NL-PL models on both code-snippets and natural language leads to state-of-the-art performance in code-related tasks [24, 31]. For our approach, the code-words are first split into subword-tokens using WordPiece tokenizer [71] (see Figure 4, bottom right). This subword tokenization is common practice in neural language models and, amongst other advantages, enables the model to process unknown words that did not occur in the training data. Subsequently, the subword-tokens are used as input to a pre-trained Transformer [24, 31] whose weights are frozen (i.e., the weights of the NL-PL model are not adjusted during the training of our architecture). This Transformer computes a high-dimensional embedding of each token that takes into account the entire input code as context. The representations of the subword-tokens are then merged to correspond to the original code-words. To reduce the

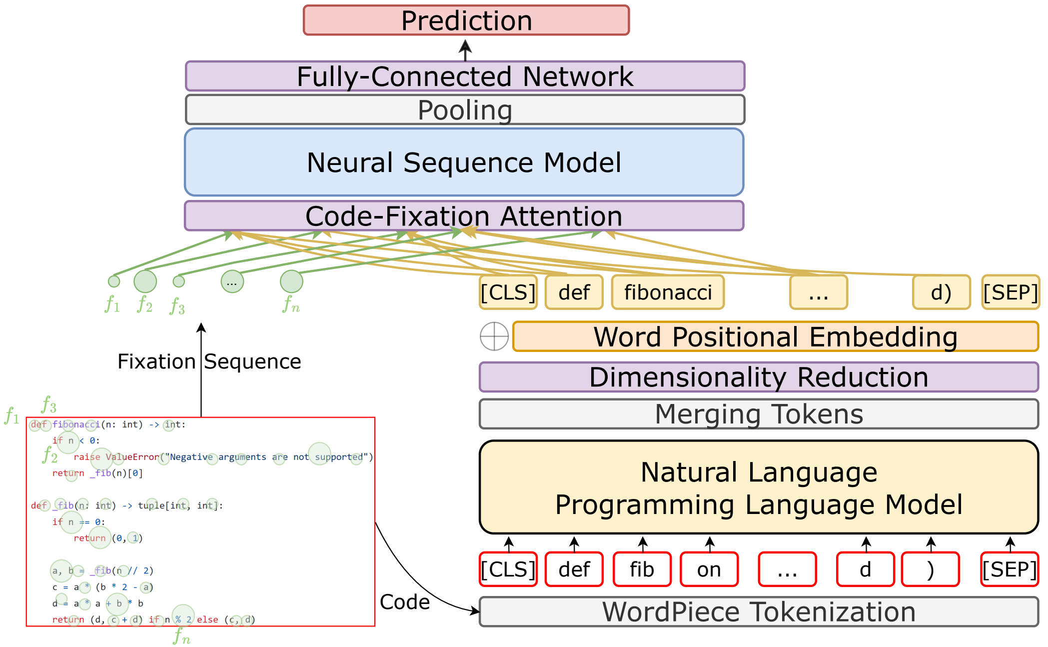


Fig. 4. Model architecture. The architecture processes both the code and the corresponding eye movement sequence. The properties of the following building blocks are treated as hyperparameters: Neural Sequence Model, Code-Fixation Attention, Word Positional Embedding, Dimensionality Reduction, Pooling and Fully Connected Network. For more details, see Section 6.1 and Table 2.

Table 2. Hyperparameters tuned during nested cross-validation.

Hyperparameter	Values
NL-PL model	GraphCodeBERT, CodeBERT
Word Positional Embedding	{yes, no}
Neural Sequence Model	LSTM, BiLSTM
Number of Sequence Layers	{1, ..., 5}
Number of Sequence Units	$\{2^i$ for $i \in [4, \ldots, 10]\}$
Bottleneck Embedding	{yes, no}
Pooling Type	{Max, Average, Last Hidden State}
Fixation-Code Attention Window	{1, 3, 5 }
Number of Fully-Connected Layers	{1, ... 5}
Number of FC Hidden Units	$\{2^i$ for $i \in [4, \ldots, 8]\}$

dimension of the embedding, we feed the code-words into a bottleneck-layer. The output of the bottleneck-layer is then our contextualized code-word representation. To account for the position of each code-word within the entire code, we add a learnable positional encoding [17].

Code-Fixation Attention. As readers do not read code linearly in the order of the code-words, but rather skip words or lines and perform regressive saccades to previously already fixated parts of the code, eye movements in code reading are characterized by two not-aligned sequence axes: the chronological axis of the fixations and the axis that defines the order of the code-words. A major architectural challenge consists of aligning these two axes without aggregating the data over one or the other axis. We resort to neural attention as a mechanism to align these two sequences. Conceptually, the idea is that we try to mimic human visual processing: Humans not only process visual information that is displayed directly on the fixation location (fovea), but also from the surrounding area (parafovea), which usually extends a couple of words to the left and the right of the fixated word. Although in this parafoveal region, vision is less acute, it has been shown that readers still extract semantic information from this region [65]. To account for this, we use windowed neural attention, where each fixation acts as a query, while the code-words are keys and values. Broadly speaking, attention learns, depending on the fixation location, how important each code-word in the attention window is for our task. The specific window-size, i.e. the code-words taken into account around the fixated word, is a hyperparameter (see Table 2).

Our representation of each fixation is the output of the neural attention mechanism concatenated with the corresponding fixation duration.

Neural Sequence Model. The neural sequence model takes as input the sequence of fixations and the code. The specific neural sequence architecture is a hyperparameter. We either use an LSTM [34] or BiLSTM [28, 66] model. LSTM and BiLSTM inherently take the sequential order of the data into account. Each architecture is comprised of several layers followed by either global max or average pooling.

Fully-Connected Neural Network. The output of the pooling layer described above is the input to a fully-connected neural network. This network consists of several fully-connected layers, each followed by a rectified linear unit (ReLU) activation function. The final output layer is a single-unit followed by a sigmoid activation function. The final output is interpreted as a binary score for code comprehension, or perceived difficulty.

6 MACHINE LEARNING EXPERIMENTS

In this section, we first provide details on the evaluation protocol before we report on the results of our experiments.

6.1 Evaluation Protocol

To evaluate the performance of our models as well as to tune our architecture, we use nested cross-validation on the collected dataset (see section 3.5). We stratify the data such that readability variants of the same code snippets are part of the same fold, and that each label class is represented at least once in the test data. We split the data into two categories: 'New Participant' and 'New Code Snippet'. In the 'New Participant' split, we ensure that training data does not contain any data from the test participants. In the 'New Code Snippet' split, we analogously ensure that none of the code snippets of the testing set is used during training. Due to the imbalance in the dataset (see table 1), we aggregated the code snippet split into 7 folds and the participants split into 4 folds to stabilize the results and reduce standard error. For the hyperparameter search, we tune our hyperparameters (see table 2) for all split and problem-setting combinations. The aggregated search results are provided in the data package [1].

Hardware and Framework. We train all neural networks using the Keras [13] and Tensorflow [16] libraries on an NVIDIA A100-SXM4-40GB GPU using the NVIDIA CUDA platform. We use the Adam optimizer.

6.2 Results

Overall, our proposed architecture achieves a new state-of-the-art performance when aligning code and fixations in a deep neural network. Table 3 provides an overview of the results.

RQ1. Our approach (bimodal) achieves an AUC of 0.746 when predicting code comprehension, and 0.739 when predicting perceived code difficulty for a participant that has not been encountered before. While the AUC of 0.743 is similar for predicting perceived code difficulty for a new code snippet, the AUC for code comprehension decreases to 0.675 showing that it is more difficult to predict code comprehension for a new code snippet than for a new participant.

Note that a leave-one-participant-out cross-validation with 27 folds instead of the 4 folds we used for stability reasons, showed that we could achieve higher performance, but also resulted in a higher standard error, leading to unstable results that are highly dependent on the characteristic of an individual fold. Using less but larger folds mitigates these effects, which is why we only report on these.

RQ2. In comparison to the three state-of-the-art methods [3, 25, 33], in almost all settings our model significantly outperforms two of the three current state-of-the-art machine learning models. For Perceived Code Difficulty, we can see that for both splits, our model outperforms the state-of-the-art methods and is significantly better than two of them. These results indicate that the stimulus code, which is only integrated in Harada and Nakayama [33] and our method, leads to major performance improvements when trying to predict perceived code difficulty.

RQ3. To examine the value of aligning fixations with code, we performed an ablation study comparing our bimodal model (using both code and fixations as input) to its ablated versions of a code-only and fixations-only model. The results show that the bimodal model outperforms its ablated versions in all splits and problem settings, except for the New Participant Split in the Code Comprehension setting, where an even mean AUC score is achieved between the bimodal and fixations-only model of our approach. While fixations appear to be the dominant contributor to performance in our approach, aligning the fixations with code into a bimodal model improves the overall performance in all splits and problem settings. Especially for the New Participant Perceived Code Difficulty evaluation, where the fixation-only model performs worse than Harada and Nakayama [33] and also exhibits a high standard error of 0.090, the bimodal model outperforms Harada and Nakayama [33] and has a reduced standard error of just 0.026. Even for the New Participant Code Comprehension evaluation, where the mean AUC performance is on par, the bimodal has a slightly lower standard error.

Note that we have included the results for the New Participant split with the code-only model for completeness reasons, but they should be interpreted with caution. Different participants that are part of training and test folds (in the New Participant split) might have worked on some of the same code comprehension tasks. Since the code-only model solely extracts information from the code snippet data and no participant-related data to make predictions, the model could potentially overfit for certain tasks.

7 DISCUSSION

We have developed and evaluated an end-to-end trained deep neural sequence model that processes both written code and a developer's eye gaze to predict code comprehension and perceived difficulty of understanding the code. Our results show that our proposed deep neural network is able to predict code comprehension with an AUC of 0.746 **(RQ1a)** and perceived code difficulty with an AUC of 0.739 **(RQ1b)**, outperforming current state-of-the-art in all settings and splits **(RQ2)** with detailed results found in Table 3. Especially for the prediction of code comprehension when split by

Table 3. AUC results ± standard error. Using a one-tailed t-test, the asterisk * indicates values that are better than random guessing (p-value < 0.05), while † indicates models that are significantly worse than best model for a setting (two-tailed t-test, p-value < 0.05). We perform nested cross-validation.

	Model	New Participant	New Code Snippet
Code Comprehension	Fritz et al. [25]	0.522±0.006*†	0.515±0.025†
	Al Madi et al. [3]	0.597±0.041†	0.549±0.014*†
	Harada et al. [33]	0.596±0.022*†	0.595±0.066*†
	Our (bimodal)	**0.746±0.021***	**0.675±0.050***
	Our (fixations)	**0.746±0.031***	0.616±0.040*
	Our (code)	0.742±0.064*	0.513±0.061†
Perceived Code Difficulty	Fritz et al. [25]	0.487±0.097†	0.461±0.028†
	Al Madi et al. [3]	0.539±0.049†	0.521±0.096†
	Harada et al. [33]	0.699±0.019*	0.632±0.115*
	Our (bimodal)	**0.739±0.026***	**0.743±0.058***
	Our (fixations)	0.690±0.090*	0.730±0.040*
	Our (code)	0.663±0.042*	0.449±0.023†

code snippet (New Code Snippet), we can see that the best baseline does not achieve a performance significantly above random guessing. This indicates that our approach using deep learning to extract features from the "raw" signals (of both stimulus code and eye movements) is beneficial. Our method incorporates recently developed NL-PL models to encode the fixations on the code. In contrast to previous approaches, our end-to-end approach overcomes the need for hand-crafted areas of interest and/or engineered eye movement features, such as mean or total fixation durations on a certain pre-defined part of the code as done by all other reference methods [3, 25, 33]. Instead, our approach takes as input the code and the chronological sequence of fixation locations and durations. The mapping of fixations to code, specifically the semantic representation computed by the NL-PL model, is achieved via a windowed neural code-fixation-attention mechanism (a detailed description can be found in section 5). In the evaluation, each parameter in the hyperparameter grid was selected at least once. In most folds, the neural code-fixation-attention mechanism utilized a window size of at least three. Additionally, the model architecture included a learnable word position embedding, with four LSTM and three fully connected layers. All other hyperparameters were approximately evenly distributed across folds.

To answer **(RQ3)** we performed an ablation study of our bimodal approach, where we have shown that a model that solely relies on code in the form of contextualized embeddings as input is not able to predict perceived code difficulty accurately and can not generalize well over new code snippets, given the amount of training data provided in this study. However, leveraging the information gain of integrating code and aligning it with fixations of the human gaze into a bimodal model results in a performance increase that outperforms the dominant fixations-only model as well as previous state-of-the-art reference models.

Note that the model's generalizability is limited regarding the code snippet selection criteria and the participants. While we put effort into the selection process to capture a diverse set of code

snippets representative of real-world code, the selected snippets are also limited due to the logistical constraints imposed by the eye-tracker and the controlled study setup. For instance, by focusing on the "vanilla" Python language and API, we capture code elements frequently used in most projects yet exclude code involving external library calls or other language features, for which further research is necessary to assess the model's applicability. Also, while the number of participants is similar to related eye-tracking studies [3, 25], it is limited based on logistical constraints (time and location) and the way we advertised, restricting the model's generalizability regarding aspects such as experience or domain knowledge. Investigating the model's performance with a larger dataset is an interesting prospect, also since neural networks generally have better performance the larger the training set. However, even with the current dataset our model outperforms the reference models and shows its ability to capture at least some human aspect.

Human-Inclusive Measure of Code Quality. While this exploratory study focused on developing a novel approach to predict the comprehension and perceived difficulty of code, it also lays the foundation for future work towards a human-inclusive quality measure of code. Evaluating the performance of a deep neural sequence model that aligns fixations and code-words on a fine-grained level has been the first step. As Peitek et al. [55] already demonstrated, popular complexity metrics fail to capture comprehension aspects that developers use when subjectively rating comprehension tasks. Our human-inclusive model goes a step further, taking into account the human gaze in predicting comprehension and perceived code difficulty. However, further studies, for example with an fMRI, are needed to examine how much of the developers' subjectivity is captured. Especially if the gaze data is automatically simulated at some point (see below) rather than tracked from the individual developer, the model might have to be adjusted for the variation in developers, such as their different levels of expertise.

To apply our approach as a human-inclusive measure of code quality directly in practice, one would require an eye-tracker to collect the eye gaze data and then map it to the code. Several approaches have already examined the value of tracking and sharing real-time eye gaze data to support scenarios such as pair programming [15], code review [12] or code tutoring [82]. With the advances in technology that allow us to use webcams for tracking eye gaze [82], such an approach might soon be feasible.

Our longer-term objective, however, is to eliminate the need for an eye-tracker to collect eye movement data as input for our model. In particular, we propose developing an eye movement generator to automatically simulate human gaze on code and provide the synthetically predicted scan paths to our model, an approach that has been looked at for natural language text [26, 58]. Augmenting our current data set could further improve the model's performance and eventually eliminate the need for human gaze as input to the model. The model would then be able to predict the perceived code difficulty based solely on code input and without the need of an eye-tracker, while still providing a human-inclusive quality measure of code. Yet, as mentioned above, one might have to account for variations in developers, such as their level of expertise in some way.

A comprehensibility measure like our perceived difficulty score could be used in continuous integration pipelines or as an educational tool to provide instant feedback for programming novices trying to optimize their code for better comprehensibility. Providing such a code quality measure that evaluates the human perception of code comprehension difficulty rather than artificially aggregating code statements could have a far-reaching impact on the industry and education by increasing efficiency, lowering entry barriers, and improving maintainability through better understandable code.

While we modeled both code comprehension and perceived code difficulty as a binary score in our study, future work should explore using a more fine-grained score (either discrete or continuous). Compared to our current binary score which serves more as an indication of problematic code,

fine-grained gradations could better reflect small changes in code quality and improve the model's applicability in practice.

More than Code. The data presented in Section 3.5 illustrates that a perceived difficulty measure reflects human code comprehension better than traditional measures like McCabe's cyclomatic complexity. Table 1 also illustrates a further weakness of the cyclomatic complexity measure: it neglects docstrings, comments, and meaningful identifiers in the score calculation. By having introduced code snippet variations with removed docstrings and obfuscated identifiers to our code snippet pool, we can observe how the cyclomatic complexity stays the same across code snippet variations, whereas the perceived difficulty changes in correlation to comprehension correctness. For example, the original code snippet *CS-03-V1* that contains docstrings, comments, and meaningful identifiers has been solved correctly by 60% of the participants with a perceived difficulty score of 2.8. However, the same code snippet but without docstrings, without comments, and with obfuscated naming (*CS-03-V3*) was only solved correctly by 8% of the participants. The performance drop is not reflected in the cyclomatic complexity, which stays the same (i.e. 4 on an open-end scale), whereas our perceived difficulty score drops from 2.8 to 1.75 (on a scale from 1 to 5; 1 being very difficult, 5 being very easy).

8 THREATS TO VALIDITY

There are several threats to the external, internal, and construct validity of our study.

External Validity. Our study was conducted with undergraduate and graduate students who had varying degrees of programming experience (from novices to programmers with eight or more years of programming experience). The experiments were conducted during all working days of the week at times ranging from morning to late afternoon. The majority of participants felt awake (only one stated to be very tired). Due to the broad range of programming experience of our participants and the well-distributed experiment times, our results provide some initial evidence of the ability to generalize to developers of different expertise and mental states. However, we do want to point out that conducting experiments with professionals could be necessary to confirm the applicability of our findings to the professional environment.

We selected the code snippets for this study in a multi-phase filtering and selection process to ensure that we select a broad range of complexities in our snippets. The comprehension tasks for the code snippets have been crafted as simple function calls to the code snippet with the intend to maximize a participant's focus on the snippet and reduce their focus on the tasks. Although we tried to minimize the task's impact and focus the participant's attention and effort on the actual snippet, we cannot be certain of its influence. We validated our snippets and tasks to minimize author bias and confirm that our snippet pool covers a broad range of complexity. Further studies are needed to investigate to what extent our findings generalize beyond individual code snippets that fit on a single screen or to code that requires substantial domain knowledge. For instance, by focusing on the "vanilla" Python language and API, we capture code elements frequently used in most projects yet exclude code involving external library calls or other language features, for which further research is necessary to assess the model's generalizability and applicability.

Internal Validity. We conducted the study in a controlled experiment setting. The experimenter followed a script to introduce and guide the participants through the experiment. The experiment started with a familiarization phase in which the participants got to know the virtual environment by completing two practice trials. Learning effects within code snippets were avoided by arranging the stimuli such that each participant saw only one version of each code snippet. This has been achieved by creating multiple random lists of code snippets with the condition that variations of code snippets are mutually exclusive across the lists. To additionally facilitate the possibility of

participants taking part in a second session, we had prepared two lists of code snippets that were mutually exclusive with the three main lists. The mutual exclusivity led to some imbalance in the dataset, where a few of the code snippets have been seen by more participants than others. While the imbalance can impact the performance of our neural network, we tried to mitigate potential effects by training and evaluating using larger folds as described in section 6.1. Furthermore, the snippets and comprehension tasks are quite distinct, reducing the possibility of learning effects. Most importantly, the comprehension questions were carefully designed such that they required in-depth code comprehension. We followed these steps and protocol to mitigate threats to internal validity.

Construct Validity. Code comprehension is a diffuse construct and researchers disagree on how studies should be designed to examine code comprehension [81]. We limited ourselves to assert code comprehension by using single-choice comprehension tasks, to avoid contamination of our eye-tracking data with comprehension-unrelated behavior. Code comprehension does not necessarily imply the ability to summarize, extend, or fix code. More research is necessary to establish a better theoretical foundation on the human comprehension process, such that standardized code comprehension methodologies can be developed and applied by studies like ours. Similar issues have been reported and summarized by Wyrich et al. [81].

To improve our construct validity, we asked our participants after each comprehension task whether they understood the code snippet. The subjective comprehension assessment of our participants was congruent with a correct comprehension task outcome in about 80% of the cases. However, subjective comprehension assessments might not be an ideal measure themselves, as participants stated that they understood the code snippet but were unable to solve the correlating task correctly in some instances.

9 CONCLUSION

This paper introduces the first end-to-end deep neural network that predicts code comprehension and perceived difficulty from a developer's eye movements during code reading. In a novel architecture that aligns code tokens with a developer's eye gaze via code-fixation attention, our approach takes advantage of both, human features extracted from developers' eye fixations and code and language features from natural-language programming-language models.

Based on a carefully designed code comprehension experiment with 27 participants working on 16 diverse tasks, we collected an eye-tracking-while-code-reading dataset. Using this dataset, we trained and assessed our developed neural network. The results of our analysis demonstrate that our bimodal approach outperforms its ablated versions (code- or fixations-only) and three state-of-the-art reference models.

The results provide evidence for the potential of a more human-inclusive and accurate model of code quality that is capable of detecting when and where a developer may have difficulty understanding code. Such a model, when integrated into the IDE or the continuous integration pipeline, could assist developers in writing better code in the first place or point out and prioritize segments for refactoring or bug fixing. At the same time, such a model could also be used to support developers by determining when a developer might not comprehend the code they are working on and maybe should take a break or ask a colleague.

Even though the current approach is based on eye movement data collected for a set of code snippets, the findings show that applying it to new code snippets (not trained on) yields good results. Approaches to generate synthetic eye movement data might even allow us to apply the model to arbitrary code snippets in the future without the need for eye-tracking, opening up further opportunities.

10 DATA AVAILABILITY

All anonymized raw and processed eye-tracking data of this study, as well as code snippets and scripts to reproduce the results are openly provided online [1].

ACKNOWLEDGEMENTS

The authors want to express their heartfelt gratitude to all the study participants whose invaluable contributions made this research possible. We also thank the anonymous reviewers for their insightful comments and suggestions. This work was partially funded by the German Federal Ministry of Education and Research under grant 01|S20043 (AEye, PI: Lena Jäger).

REFERENCES

[1] 2024. Supplementary Material. (2024). https://zenodo.org/doi/10.5281/zenodo.11123100
[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge.
[3] Naser Al Madi, Cole S. Peterson, Bonita Sharif, and Jonathan I. Maletic. 2021. From Novice to Expert: Analysis of Token Level Effects in a Longitudinal Eye Tracking Study. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 172–183. https://doi.org/10.1109/ICPC52881.2021.00025
[4] Salwa Aljehane, Bonita Sharif, and Jonathan Maletic. 2021. Determining Differences in Reading Behavior between Experts and Novices by Investigating Eye Movement on Source Code Constructs during a Bug Fixing Task. In *Eye Tracking Research and Applications Symposium (ETRA)*, Vol. PartF169257. Association for Computing Machinery. https://doi.org/10.1145/3448018.3457424
[5] Roman Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies* 70, 2 (2012), 143–155. https://doi.org/10.1016/j.ijhcs.2011.09.003
[6] Tanya Beelders and Jean Pierre Du Plessis. 2016. The influence of syntax highlighting on scanning and reading behaviour for source code. In *ACM International Conference Proceeding Series*, Vol. 26-28-September-2016. Association for Computing Machinery. https://doi.org/10.1145/2987491.2987536
[7] Charles Bigelow. 2019. Typeface features and legibility research. *Vision Research* 165 (12 2019), 162–172. https://doi.org/10.1016/j.visres.2019.05.003
[8] Barry W. Boehm. 1981. *Software engineering economics.* Prentice-Hall.
[9] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (6 1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5
[10] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *IEEE International Conference on Program Comprehension*, Vol. 2015-August. IEEE Computer Society, 255–265. https://doi.org/10.1109/ICPC.2015.36
[11] G. Ann Campbell. 2017. *Cognitive Complexity - A new way of measuring understandability.* Technical Report. SonarSource SA, Switzerland. https://www.sonarsource.com/docs/CognitiveComplexity.pdf
[12] Shiwei Cheng, Jialing Wang, Xiaoquan Shen, Yijian Chen, and Anind Dey. 2021. Collaborative eye tracking based code review through real-time shared gaze visualization. *Frontiers of Computer Science* 16, 3 (Nov. 2021). https://doi.org/10.1007/s11704-020-0422-1
[13] François Chollet et al. 2015. *Keras.* https://keras.io
[14] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings.* Association for Computing Machinery, Inc, 107–118. https://doi.org/10.1145/2786805.2786838
[15] Sarah D'Angelo and Andrew Begel. 2017. Improving Communication Between Pair Programmers Using Shared Gaze Awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6245–6290. https://doi.org/10.1145/3025453.3025573
[16] Jeffrey Dean, Rajat Monga, et al. 2015. *TensorFlow: Large-scale machine learning on heterogeneous systems.* https://www.tensorflow.org/
[17] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019), 4171–4186. https://doi.org/10.18653/v1/N19-1423
[18] William H Dubay. 2004. The Principles of Readability. (2004).

[19] Moshe Eizenman, P.E Hallett, and R.C. Frecker. 1985. Power spectra for ocular drift and tremor. *Vision Research* 25, 11 (1985), 1635–1640. https://doi.org/10.1016/0042-6989(85)90134-8

[20] Onyeka Ezenwoye. 2018. What language?-The choice of an introductory programming language. In *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8. https://doi.org/10.1109/FIE.2018.8658592

[21] Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2020. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering* 25, 3 (May 2020), 2140–2178. https://doi.org/10.1007/s10664-019-09751-4

[22] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 73–82. https://doi.org/10.1109/ICPC.2012.6240511 ISSN: 1092-8138.

[23] Dror G Feitelson. 2021. Considerations and pitfalls in controlled experiments on code comprehension. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 106–117. https://doi.org/10.1109/ICPC52881.2021.00019

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics Findings of ACL: EMNLP 2020* (2 2020), 1536–1547. https://doi.org/10.48550/arxiv.2002.08155

[25] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings - International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014, 1)*. IEEE Computer Society, 402–413. https://doi.org/10.1145/2568225.2568266

[26] Wolfgang Fuhl and Enkelejda Kasneci. 2018. Eye movement velocity and gaze data generator for evaluation, robustness testing and assess of eye tracking software and visualization tools. *arXiv preprint arXiv:1808.09296* (2018).

[27] Simone Grassini, Giulia Virginia Segurini, and Mika Koivisto. 2022. Watching Nature Videos Promotes Physiological Restoration: Evidence From the Modulation of Alpha Waves in Electroencephalography. *Frontiers in Psychology* 13 (6 2022). https://doi.org/10.3389/fpsyg.2022.871143

[28] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. 2005. Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition. In *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005*, Włodzisław Duch, Janusz Kacprzyk, Erkki Oja, and Sławomir Zadrożny (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 799–804. https://doi.org/10.1007/11550907_126

[29] David Grimes, Desney S Tan, Scott E Hudson, Pradeep Shenoy, and Rajesh P N Rao. 2008. Feasibility and Pragmatics of Classifying Working Memory Load with an Electroencephalograph. *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08* (2008). https://doi.org/10.1145/1357054

[30] Kristoffer Gunnarsson and Olivia Herber. 2020. *The Most Popular Programming Languages of GitHub's Trending Repositories.* Technical Report.

[31] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. (9 2020). https://doi.org/10.48550/arxiv.2009.08366

[32] Eija Haapalainen, Seungjun Kim, Jodi F Forlizzi, and Anind K Dey. 2010. Psycho-Physiological Measures for Assessing Cognitive Load. *Proceedings of the 12th ACM international conference on Ubiquitous computing* (2010). https://doi.org/10.1145/1864349

[33] Hiroto Harada and Minoru Nakayama. 2021. Estimation of Reading Ability of Program Codes Using Features of Eye Movements. In *ACM Symposium on Eye Tracking Research and Applications* (Virtual Event, Germany) *(ETRA '21 Short Papers)*. Association for Computing Machinery, New York, NY, USA, Article 32, 5 pages. https://doi.org/10.1145/3448018.3457421

[34] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[35] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. 2011. *Eye tracking: A comprehensive guide to methods and measures.* Oxford University Press, Oxford.

[36] Shamsi T Iqbal, Sam Zheng, and Brian P Bailey. 2004. Task-Evoked Pupillary Response to Mental Workload in Human-Computer Interaction. *Extended abstracts of the 2004 conference on Human factors and computing systems - CHI '04* (2004), 1477–1480. https://doi.org/10.1145/985921

[37] Toyomi Ishida and Hidetake Uwano. 2019. Synchronized analysis of eye movement and EEG during program comprehension. In *Proceedings - 2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming, EMIP 2019*. Institute of Electrical and Electronics Engineers Inc., 26–32. https://doi.org/10.1109/EMIP.2019.00012

[38] Marcel Adam Just and Patricia A. Carpenter. 1980. A theory of reading: from eye fixations to comprehension. *Psychological review* 87 4 (1980), 329–54. https://doi.org/10.1037/0033-295X.87.4.329
[39] Michele Lacchia. 2022. Radon 4.1.0 documentation — Using radon programmatically. https://radon.readthedocs.io/en/latest/api.html
[40] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05* (2005). https://doi.org/10.1145/1067445
[41] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. 2018. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing* (2018). https://doi.org/10.1007/s10586-017-0746-2
[42] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro Interaction Metrics for Defect Prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 311–321. https://doi.org/10.1145/2025113.2025156
[43] Meir M Lehman. 1980. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1980), 213–221. https://doi.org/10.1016/0164-1212(79)90022-0
[44] H. F. Li and W. K. Cheung. 1987. An Empirical Study of Software Metrics. *IEEE Transactions on Software Engineering* SE-13, 6 (1987), 697–708. https://doi.org/10.1109/TSE.1987.233475
[45] Radu Marinescu. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE, 350–359. https://doi.org/10.1109/ICSM.2004.1357820
[46] Anneliese Von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076
[47] Thomas J. Mccabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837
[48] Steve McConnell. 2004. *Code complete*. Pearson.
[49] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50
[50] Sebastian C. Müller and Thomas Fritz. 2016. Using (Bio)Metrics to Predict Code Quality Online. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 452–463. https://doi.org/10.1145/2884781.2884803
[51] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th international conference on Software engineering*. 284–292. https://doi.org/10.1109/ICSE.2005.1553571
[52] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th international conference on Software engineering*. 452–461. https://doi.org/10.1145/1134285.1134349
[53] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering* 24, 3 (June 2019), 1418–1457. https://doi.org/10.1007/s10664-018-9664-z
[54] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 348–359. https://doi.org/10.1109/ICSME46990.2020.00041
[55] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 524–536. https://doi.org/10.1109/ICSE43902.2021.00056
[56] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. 2022. Correlates of Programmer Efficacy and Their Link to Experience: A Combined EEG and Eye-Tracking Study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 120–131. https://doi.org/10.1145/3540250.3549084
[57] Nancy Pennington. 1987. Comprehension Strategies in Programming. In *Empirical Studies of Programmers: Second Workshop*. Ablex Publishing Corp., 100–113. https://doi.org/10.5555/54968.54975
[58] Paul Prasse, David Robert Reich, Silvia Makowski, Seoyoung Ahn, Tobias Scheffer, and Lena A. Jäger. 2023. SP-EyeGAN: Generating Synthetic Eye Movement Data with Generative Adversarial Networks. In *2023 Symposium on Eye Tracking Research and Applications*. ACM, Tubingen Germany, 1–9. https://doi.org/10.1145/3588015.3588410
[59] Keith Rayner. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124, 3 (Nov. 1998), 372–422. https://doi.org/10.1037/0033-2909.124.3.372
[60] SR Research. 2023. https://www.sr-research.com/eyelink-portable-duo/. Retrieved Feb 1, 2023.
[61] Kilseop Ryu and Rohae Myung. 2005. Evaluation of mental workload with a combined measure based on physiological indices during a dual task of tracking and mental arithmetic. *International Journal of Industrial Ergonomics* 35, 11 (11

2005), 991–1009. https://doi.org/10.1016/j.ergon.2005.04.005

[62] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 564–575. https://doi.org/10.1145/2635868.2635895

[63] Sam. 2020. What determines whether a validation is "GOOD", "FAIR" or "POOR"? https://www.sr-research.com/support/thread-244.html

[64] Lisa Schibelius, Amanda Ross, and Andrew Katz. 2022. An Empirical Study of Programming Languages Specified in Engineering Job Postings. In *ASEE Annual Conference & Exposition*. https://doi.org/10.18260/1-2--41235

[65] Elizabeth R Schotter, Bernhard Angele, and Keith Rayner. 2012. Parafoveal processing in reading. *Attention, perception psychophysics* 74 (2012), 5–35. https://doi.org/10.3758/s13414-011-0219-2

[66] Mike Schuster and Kuldip Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (12 1997), 2673–2681. https://doi.org/10.1109/78.650093

[67] Todd Sedano. 2016. Code Readability Testing, an Empirical Study. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. 111–117. https://doi.org/10.1109/CSEET.2016.36 ISSN: 2377-570X.

[68] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. 2012. An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects. *Proceedings of the Symposium on Eye Tracking Research and Applications* (2012), 381–384. https://doi.org/10.1145/2168556

[69] Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences* 8, 3 (1979). https://doi.org/10.1007/BF00977789

[70] Elliot Soloway, Beth Adelson, and Kate Ehrlich. 1988. Knowledge and processes in the comprehension of computer programs. In *The nature of expertise*. Lawrence Erlbaum Associates, Inc, Hillsdale, NJ, US, 129–152.

[71] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. 2021. Fast WordPiece Tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 2089–2103. https://doi.org/10.18653/v1/2021.emnlp-main.160

[72] SR Research Ltd. 2016. *EyeLink ® Portable Duo User Manual EyeLink ® SR Research*.

[73] SR Research Ltd. 2020. SR Research Experiment Builder.

[74] TIOBE Software BV. 2022. TIOBE Index - TIOBE. https://www.tiobe.com/tiobe-index/

[75] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code. *Eye Tracking Research and Applications Symposium (ETRA)* (2014), 231–234. https://doi.org/10.1145/2578153.2578218

[76] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-Ichi Matsumoto. 2006. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06* (2006), 133–140. https://doi.org/10.1145/1117309

[77] Wade M Vagias. 2006. *Likert-Type Scale Response Anchors Level of Agreement*. Technical Report. Clemson International Institute for Tourism & Research Development, Department of Parks, Recreation and Tourism Management. Clemson University.

[78] Roger P. G. van Gompel, Martin H. Fischer, Wayne S. Murray, and Robin L. Hill (Eds.). 2007. *Eye movements: A window on mind and brain*. Elsevier.

[79] Talita Vieira Ribeiro and Guilherme Travassos. 2017. *Who is Right? Evaluating Empirical Contradictions in the Readability and Comprehensibility of Source Code*. Technical Report.

[80] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06)*. Australian Computer Society, Inc., 243–252. https://doi.org/10292/15405

[81] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Comput. Surv.* 56, 4, Article 106 (2023), 42 pages. https://doi.org/10.1145/3626522

[82] Stephanie Yang, Amreen Amin Poonawala, Tian-Shun Allan Jiang, and Bertrand Schneider. 2023. Can Synchronous Code Editing and Awareness Tools Support Remote Tutoring? Effects on Learning and Teaching. *Proc. ACM Hum.-Comput. Interact.* 7, CSCW2, Article 328 (oct 2023), 30 pages. https://doi.org/10.1145/3610177

[83] Hongyu Zhang, Xiuzhen Zhang, and Ming Gu. 2007. Predicting Defective Software Components from Code Complexity Measures. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. 93–96. https://doi.org/10.1109/PRDC.2007.28

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