EYEMULATOR: Improving Code Language Models by Mimicking Human Visual Attention

Anonymous ACL submission

Abstract

Code Language Models (CodeLLMs) traditionally learn attention based solely on statistical input-output token correlations ("machine attention"). In contrast, human developers rely on intuition, selectively fixating on semantically salient tokens during program comprehension. We present EYEMULATOR, a model-agnostic technique to align CodeLLM attention with human visual attention without architectural changes. By extracting scan paths from eye-tracking data, we derive token-level attention weights used to augment the loss function during fine-tuning. This induces the model to mimic human focus. Our evaluation across StarCoder, Llama-3.2, and DeepSeek-Coder shows that EYEMULATOR significantly outperforms baselines, achieving gains of over 30 CodeBLEU points in translation and up to 22 BERTScore points in summarization. Ablation studies confirm that these gains stem directly from replicating human attention dynamics. Artifacts are available at https://zenodo.org/records/16134801.

1 Introduction

Large Language Models (LLMs) have fundamentally altered the landscape of software engineering, demonstrating exceptional capability in tasks ranging from automated code generation to bug localization. These models, typically based on the Transformer architecture, learn to predict tokens by minimizing loss over internet-scale repositories (Vaswani et al., 2017). However, this process relies on "machine attention," a statistical mechanism that treats context uniformly based on data correlations. In contrast, human developers employ distinct cognitive strategies characterized by program comprehension (O'Brien, 2003; Harth and Dugerdil, 2017). Eye-tracking research consistently demonstrates that experts rely on intuition to form mental models. They fixate selectively on semantically critical elements, such as control flow conditions and method signatures, while skimming over syntactic sugar and boilerplate (Sharafi et al., 2020; Huang et al., 2020).

The dominant paradigm for adapting CodeLLMs relies heavily on minimizing prediction error over vast datasets or aligning models via high-level instruction tuning. While these methods improve general adherence to user intent, they do not fundamentally alter the model's underlying attention mechanism, which remains tethered to statistical co-occurrences rather than semantic reasoning. Existing attempts to bridge this gap, such as retrieval-augmented generation (RAG), provide external context but fail to teach the model *how* to process that context like an expert. Consequently, even state-of-the-art models continue to distribute attention uniformly across boilerplate and logic, missing the nuanced, selective focus that characterizes human program comprehension.

To bridge the critical gap between statistical machine attention and selective human focus, we propose EYEMULATOR, a methodology to ground CodeLLM training in cognitive visual patterns. We posit that while purely data-driven models excel at surface-level pattern matching, they fundamentally lack the ability to prioritize the logical dependencies, such as variable data flow and execution paths, that characterize expert comprehension. By explicitly aligning model attention with human scan paths, we enable the system to process code with a semantic salience mirroring that of a developer. Crucially, unlike prior gaze-aware approaches that necessitate specialized architectures or expensive pre-training from scratch (Zhang et al., 2024), we aim to distill these cognitive insights into a modular, model-agnostic signal that can be seamlessly integrated into standard supervised fine-tuning pipelines.

The core innovation of EYEMULATOR is a generative attention mechanism that allows standard

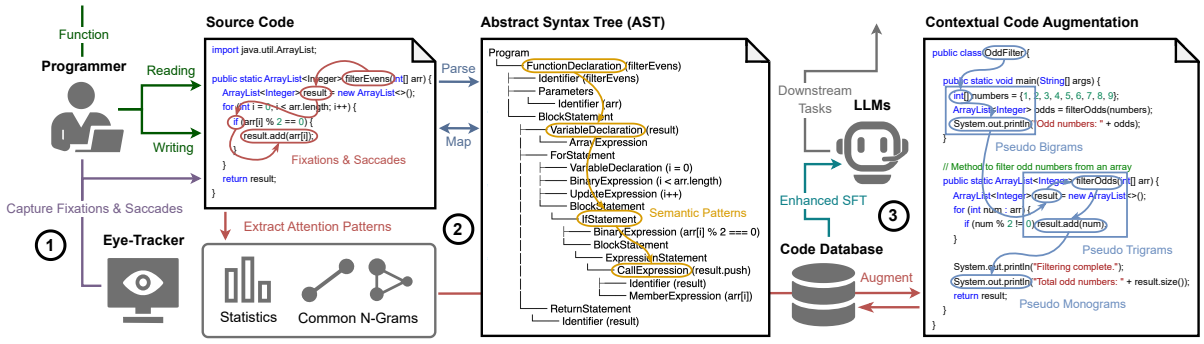

Figure 1: Overview of the EYEMULATOR framework. The pipeline begins with ① **Gaze Data Acquisition** from human developers. This data is processed in ② **Artifact Distillation**, where fixations are mapped to AST tokens to derive Semantic Salience Priors (static importance) and Transition Probabilities (sequential flow). Finally, in ③ **Gaze-Informed Fine-Tuning**, these artifacts generate pseudo-scan paths that guide the model using a dual-objective loss (Weighted SFT + DPO), aligning machine attention with human cognitive patterns.

CodeLLMs to learn human cognitive patterns directly from text. We achieve this by first distilling raw eye-tracking data into two portable artifacts: *Semantic Salience Priors*, which model the static importance of token types (e.g., variables vs. keywords), and *Transition Probabilities*, which capture the dynamic flow of expert reading (e.g., jumping from definition to usage). These artifacts enable us to synthesize "pseudo-scan paths" for standard training data where no eye-tracking exists. We then align the model to these paths using a composite objective: a re-weighted Supervised Fine-Tuning (SFT) loss to emphasize salient tokens, and a token-level Direct Preference Optimization (DPO) loss that explicitly rewards the model for prioritizing human-preferred context over irrelevant boilerplate.

We evaluate EYEMULATOR on three diverse tasks from the CodeXGlue benchmark, covering code translation, summarization, and completion. To ensure robustness, we test our approach across three distinct backbone architectures: StarCoder (1B), Llama-3.2 (1B), and DeepSeek-Coder (1.3B). Our results demonstrate that injecting human attention priors yields substantial performance improvements across all settings. Specifically, EYEMULATOR surpasses state-of-the-art baselines by up to 34.35 points in CodeBLEU for translation tasks and 22.92 points in BERTScore for summarization tasks. Furthermore, extensive ablation studies and qualitative analysis of attention maps confirm that these gains are directly attributable to the model adopting sharper, more human-aligned attention distributions rather than simply overfitting to the dataset.

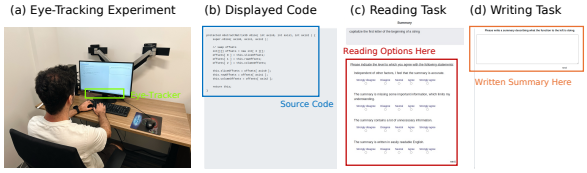

Figure 2: EyeTrans dataset overview. Gaze recordings (a) on Java snippets (b) are processed to align fixations with ASTs, separated into Reading (c) and Writing (d) sessions.

2 Approach

As illustrated in Figure 1, EYEMULATOR consists of three stages: processing raw gaze data, extracting statistical attention priors, and fine-tuning the model using a dual-objective loss.

2.1 Data Transformation and Metrics

We leverage the EyeTrans corpus (Zhang et al., 2024), which contains 120Hz gaze recordings from 27 programmers reading and writing Java code. To transform raw sensor data into a training signal, we first apply a dispersion-threshold (I-DT) algorithm to identify **Fixations** ($\mathcal{F}$), defined as stable gaze points lasting at least 100 ms. These fixations serve as our primary metric of cognitive interest. We then spatially map these fixations to Abstract Syntax Tree (AST) leaf tokens using bounding boxes captured during the study (Figure 2). Unmapped points, which account for approximately 3% of the data, are discarded. This process yields 1,565 **Scan Paths** ($\mathcal{P}$), representing the chronological sequence of attended tokens aligned with the code structure.

2.2 Attention Pattern Extraction

We distill the token-aligned fixations into two statistical artifacts: semantic salience priors and sequential transition models.

Semantic Salience Priors. To model the inherent importance of different token types (e.g., `ForStatement` vs. `Identifier`), we employ a Bayesian approach. For each semantic class s, we count the number of fixations $c_1(s)$ relative to the total number of available tokens n_s^{tok}. We model the probability of attention θ_s using a Beta distribution $\text{Beta}(\alpha_s, \beta_s)$, which serves as a conjugate prior to the binomial likelihood of fixation. We set $\alpha_s = c_1(s) + 1$ and $\beta_s = n_s^{\text{tok}} - c_1(s) + 1$. The posterior mean $\mathbb{E}[\theta_s] = \alpha_s/(\alpha_s + \beta_s)$ provides a robust estimate of salience, smoothing out noise in rare token classes.

Sequential Transition Models. To capture the flow of expert reading, we count bigrams and trigrams within the scan paths. We compute conditional probabilities $P(s_b|s_a)$ and $P(s_c|s_a, s_b)$, which represent the likelihood of transitioning between specific semantic states (e.g., from a variable definition to its usage). We prune n-grams with fewer than 5 occurrences to reduce noise.

2.3 Pseudo-Attention Path Generation

To simulate human attention for standard training data where no eye-tracking exists, we generate "pseudo scan paths" $\tilde{\mathcal{P}}$. For an input sequence of length n, we proceed in three steps: (1) We sample a saliency ratio ρ from the aggregate Beta distribution of the corpus. (2) We determine the total count of attended tokens $m = \lfloor \rho n \rfloor$ and allocate quotas to specific token classes based on their posterior means $\mathbb{E}[\theta_s]$. (3) We construct the path $\tilde{\mathcal{P}}$ by greedily matching masked tokens into valid n-grams (prioritizing Trigrams, then Bigrams, then Monograms) that satisfy a line-span constraint L. This procedure (Figure 3) synthesizes sequences that preserve both the local structure and the semantic selectivity of human attention.

2.4 Gaze-Informed Fine-Tuning

We seamlessly integrate these artifacts into standard LLM training using a novel weighting scheme and a composite loss function.

Weight Calculation. Since LLMs typically use subword tokenizers (e.g., Byte-Pair Encoding), we project our AST-level weights by assigning the parent token's weight to all its constituent subword shards. We compute a final training weight w_j for each token j by combining a base importance term, an inverse frequency term to upweight rare code constructs, and the semantic probability: $w_j = w_{\text{base}} + (\log(\text{freq}(g_j) + 2))^{-1} + \mathbb{E}[\theta_{s_j}]$.

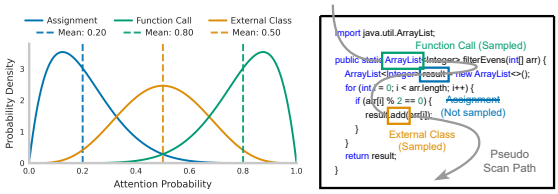

Figure 3: Pseudo-attention path generation. Salient tokens are selected via Beta priors and connected into a coherent path.

Composite Objective. We fine-tune the model using a loss function $\mathcal{L}(\phi) = \mathcal{L}_{\text{SFT}}(\phi) + \gamma\mathcal{L}_{\text{DPO}}(\phi)$. The **Weighted SFT Loss** is a modification of the standard categorical cross-entropy, scaled by the calculated weights: $\mathcal{L}_{\text{SFT}}(\phi) = -\sum_j w_j \log P_\phi(x_j|x_{<j})$. The **Token-Level DPO Loss** adapts the Direct Preference Optimization framework (Rafailov et al., 2024) to our setting. DPO typically optimizes a policy to prefer a winning sample y_w over a losing sample y_l. We define our "winning" trajectory as the generated pseudo-scan path (high salience) and the "losing" trajectory as the complement (low salience background tokens). This term explicitly rewards the model for assigning higher probability to the semantically salient tokens that humans prioritize.

2.5 Integrated Training Procedure

We synthesize the artifact extraction, path generation, and loss computation into a unified training loop. Algorithm 1 details the complete fine-tuning procedure.

The process begins by initializing the model ϕ and loading the distilled EyeTrans artifacts (Semantic Priors $\mathbb{E}[\theta]$ and Transition Probabilities P_{trans}). For every batch of code sequences, we dynamically generate pseudo-scan paths $\tilde{\mathcal{P}}$ that mimic human visual attention. These paths serve as the ground truth for calculating token-specific importance weights W. Finally, the model parameters are updated via gradient descent on the composite objective $\mathcal{L}_{\text{total}}$, which balances the reconstruction of salient tokens ($\mathcal{L}_{\text{SFT}}$) with the preference alignment of attention trajectories ($\mathcal{L}_{\text{DPO}}$).

Algorithm 1 EYEMULATOR Integrated Fine-Tuning

Input: Pre-trained CodeLLM ϕ_0, Code Dataset $\mathcal{D} = \{x^{(i)}\}_{i=1}^{N}$, EyeTrans Artifacts (Semantic Priors $\mathbb{E}[\theta]$, Transitions P_{trans})
Hyperparameters: DPO weight γ, Learning rate η
Output: Fine-tuned Model ϕ^*

$\phi \leftarrow \phi_0$
while not converged **do**
  Sample batch $\mathcal{B} \sim \mathcal{D}$
  for all sequence $x \in \mathcal{B}$ **do**
    // Stage 1: Pseudo-Path Generation (Sec. 2.2)
    $Tags \leftarrow \text{GetASTTags}(x)$
    $\rho \sim \text{Beta}(\alpha_{\text{agg}}, \beta_{\text{agg}})$ {Sample attention density}
    $\tilde{\mathcal{P}} \leftarrow \text{GeneratePath}(x, Tags, \rho, \mathbb{E}[\theta], P_{\text{trans}})$
    // Stage 2: Weight Calculation (Sec. 2.4)
    for all token $x_j \in x$ **do**
      $w_j \leftarrow w_{\text{base}} + \frac{1}{\log(\text{freq}(x_j)+2)} + \mathbb{E}[\theta_{\text{tag}(x_j)}]$
    end for
    $W \leftarrow \{w_j\}_{j=1}^{|x|}$
    // Stage 3: Dual-Objective Optimization
    $\mathcal{L}_{\text{SFT}} \leftarrow -\sum_{j \in \tilde{\mathcal{P}}} w_j \log P_\phi(x_j | x_{<j})$
    $\mathcal{L}_{\text{DPO}} \leftarrow \text{DPOLoss}(\phi, \tilde{\mathcal{P}}, x \setminus \tilde{\mathcal{P}}, W)$
    $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{SFT}} + \gamma \mathcal{L}_{\text{DPO}}$
  end for
  $\phi \leftarrow \phi - \eta \nabla_\phi \sum_{x \in \mathcal{B}} \mathcal{L}_{\text{total}}$
end while
return ϕ

3 Experimental Setup

We design our experiments to evaluate the efficacy of gaze-informed training across different code intelligence tasks and model architectures. We specifically address three research questions: (RQ1) Does mimicking human attention improve performance on downstream tasks? (RQ2) How do different components of the EYEMULATOR framework contribute to these gains? (RQ3) Does the model actually learn to attend to semantically salient regions?

Datasets and Benchmarks. We utilize the EyeTrans corpus (Zhang et al., 2024) solely for extracting attention artifacts as detailed in Section 2. For downstream evaluation, we employ the CodeXGlue benchmark (Lu et al., 2021), selecting three distinct tasks to test generalization. For *Code Translation* (Java to C#), we use the standard split of 10,300 training and 500 test pairs, measuring performance using CodeBLEU, CrystalBLEU, and Exact Match. For *Code Summarization* (Java to English), we use a subset of 16,500 examples for training and report BERTScore to capture semantic similarity alongside ROUGE-L. Finally, for *Code Completion*, we sample 20% of the CodeXGlue Java corpus (approx. 1.6k files) for token-level completion tasks, evaluating with Exact Match and CodeBLEU. To ensure reproducibility and provide a formal basis for our comparative analysis, we provide complete mathematical formulations and justifications for all performance and attention-based metrics in Appendix A.

Models and Baselines. To demonstrate model agnosticism, we apply EYEMULATOR to three state-of-the-art foundation models with varying architectures: StarCoder (1B), a dense decoder-only model; Llama-3.2 (1B), a highly optimized instruction-tuned model; and DeepSeek-Coder (1.3B), a model pre-trained with a fill-in-the-middle objective. We compare our approach against two primary baselines: *Standard SFT*, where models are fine-tuned on the same data without gaze weights, and *Random Attention*, where attention weights are assigned randomly to strictly control for the regularization effects of the weighting scheme.

Implementation Details. All models are trained using full fine-tuning rather than parameter-efficient adapters, ensuring human attention priors are deeply internalized by the model backbones. We utilize the HuggingFace `transformers` library and a custom `LlamaForCausalLM` class to integrate token-level fixation weights into the loss function. Input sequences are processed with a maximum length of 1024 tokens. Optimization is performed using the AdamW optimizer with a linear scheduler and a constant learning rate of 2×10^{-5}. We use an effective batch size of 16, and the DPO weight γ is set to 0.1. Training is conducted on a single NVIDIA L40S GPU for 3 epochs to ensure convergence while preventing overfitting. Detailed training configurations and attention-prior alignment steps are documented in Appendix B.

4 Result Analysis

We present our findings organized by the research questions proposed in Section 3. To explore how human-attention priors influence model behavior, we first compare EYEMULATOR against strong baselines on three core tasks: Java to C# translation, code completion, and summarization. We then perform ablation studies, disabling one gaze-derived module at a time to quantify its individual contribution. Finally, we analyze qualitative attention-map visualizations to examine how our approach steers the model toward semantically rich regions of code.

Table 1: Representative gaze patterns. Transitions like "conditional → loop" reveal structured reading.

Type	Sequence	Count
Mono	variable declaration	18665
Mono	conditional statement	13222
Bigram	function decl → variable decl	8399
Bigram	conditional statement → loop	6026
Trigram	func decl → param → var decl	1634
Trigram	conditional → func decl → param	1199

4.1 RQ1: Artifact Distillation

We assess how well distilled artifacts capture human attention by examining n-gram fixation patterns, fitted Beta parameters for semantic labels, and the resulting attention distributions across reading, writing, and combined tasks.

N-gram Analysis. To assess the consistency and structure of human attention, we mapped programmers' fixation sequences to semantic categories and extracted the most frequent transitions. As shown in Table 1, the resulting patterns reveal a non-linear yet highly organized reading strategy. The high count of *function declaration → variable declaration* bigrams (8,399) indicates that developers frequently switch focus between interface definitions and their implementation details. Similarly, the *conditional statement → loop* pattern (6,026) reflects an iterative inspection of branching logic and control flow. These recurring transition motifs demonstrate that expert attention is driven by logical dependencies rather than linear scanning.

Beta Parameter Estimation. Figure 4 presents the fitted Beta parameters (α_s = gaze hits, β_s = gaze misses) for each semantic label. Consistent with the n-gram findings, `variable declaration` exhibits the highest α_s in both reading and combined tasks, reinforcing its role as a primary cognitive anchor for developers. In contrast, control-flow labels like `loop` show more balanced ratios ($\alpha_s = 7,876$, $\beta_s = 4,876$), indicating that attention to these constructs is more context-dependent, shifting between cursory scanning and deeper inspection depending on the complexity of the logic.

Density Function Visualization. Figure 5 illustrates the Beta distribution density functions derived from these parameters. Reading-only curves are sharply peaked; for instance, `variable declaration` centers tightly around a high probability, signifying focused and reliable inspection. Conversely, the distributions for `conditional statement` in the combined task are broader and even bimodal. This variance suggests divergent reading strategies, where developers may either skim standard conditions quickly or fixate heavily on complex predicates, confirming that our artifacts capture the nuance of cognitive load.

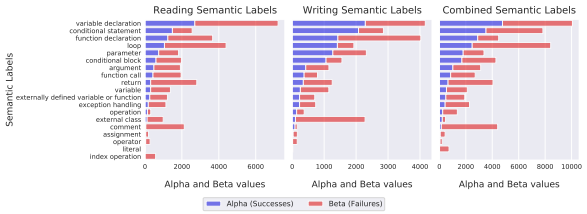

Figure 4: Estimated Beta parameters. High α_s for "variable declaration" indicates consistent attention.

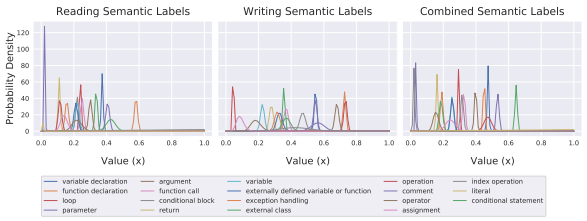

Figure 5: Smoothed Beta density functions showing focused attention for declarations vs. varied attention for control flow.

4.2 RQ2: Cross-Task Evaluation

To test whether gaze-derived priors generalize beyond summarization, we injected EyeMulator into three code-intelligence pipelines, including completion, translation, and summarization, using StarCoder, Llama-3.2, and DeepSeek-Coder as representative baselines. Table 2 reports performance improvements over each native model, demonstrating consistent gains across tasks and architectures.

Summarization Performance. Incorporating EyeMulator improves StarCoder's BERTScore from 34.04 to 51.06, a 17-point gain that yields more coherent and contextually accurate summaries. Llama-3.2 and DeepSeek-Coder see similar uplifts (+16 and +22 points respectively), producing abstracts that better emphasize method signatures, branch conditions, and variable roles. Qualitative analysis confirms that attention priors guide these models to focus on semantically critical tokens, resulting in concise yet comprehensive descriptions.

Table 2: Cross-task evaluation across three diverse architectures. EYEMULATOR (gray) consistently outperforms native baselines, achieving peak gains of +43 points in Translation (H-Exact) and +41 points in Completion (CodeBLEU). Absolute improvements are shown in parentheses.

Model	Completion			Translation			Summarization			
	CodeBLEU	Cry.BLEU	H-Exact	CodeBLEU	Cry.BLEU	H-Exact	METEOR	R-1	R-L	BERT
StarCoder	13.90	19.41	47.66	52.07	65.07	21.11	29.06	27.47	20.78	34.04
EYEMULATOR	51.98 (+38)	57.53 (+38)	79.90 (+32)	86.42 (+34)	92.61 (+27)	64.97 (+43)	33.41 (+4)	46.27 (+18)	41.09 (+20)	51.06 (+17)
Llama-3.2	19.45	26.65	50.85	71.88	82.29	53.25	27.05	24.86	18.27	33.41
EYEMULATOR	60.47 (+41)	65.90 (+39)	77.96 (+27)	83.25 (+11)	91.52 (+9)	61.02 (+7)	30.69 (+3)	44.06 (+19)	38.84 (+20)	49.49 (+16)
DeepSeek	13.80	18.98	49.40	58.69	69.94	24.13	22.81	21.96	15.12	28.64
EYEMULATOR	48.82 (+35)	54.63 (+35)	78.91 (+29)	86.34 (+27)	93.09 (+23)	65.66 (+41)	33.92 (+11)	46.86 (+24)	41.68 (+26)	51.56 (+22)

Translation and Completion. Applying the same priors to Java-C# translation and code completion delivers strong improvements. StarCoder's CodeBLEU for translation rises from 52.07 to 86.42 (+34), while its Hybrid-Exact match for completion climbs from 47.66 to 79.90 (+32). Llama-3.2 and DeepSeek-Coder exhibit comparable gains; for instance, Llama-3.2's CodeBLEU in completion jumps by over 40 points (19.45 to 60.47). These results indicate that human-attention signals enhance both fluency and correctness in generation tasks.

Architectural Robustness. The benefits of EYEMULATOR hold across three heterogeneous transformer backbones: GPT-2-based StarCoder, decoder-only Llama-3.2, and DeepSeek-Coder, which utilizes a mixture-of-experts paradigm. No changes to network architectures or extra fine-tuning were required, underlining EYEMULATOR's plug-and-play nature. This model-agnostic success highlights the versatility and scalability of distilled gaze artifacts as a lightweight mechanism for guiding attention in diverse code-intelligence settings.

4.3 RQ3: Session-Mode Analysis

Figure 6 illustrates that the proportions of semantic categories differ markedly between tasks, motivating a separate evaluation of reading-derived (EYEMULATOR(R)) versus writing-derived (EYEMULATOR(W)) priors. Table 3 breaks down performance by subcategory.

Completion Subcategories. Writing-derived artifacts (EYEMULATOR(W)) deliver the strongest gains on code completion, significantly outperforming the reading variant. For *Structural Boilerplate*, the Hybrid-Exact score rises from a baseline of 54.61 to 86.07 with writing priors, compared to just 50.22 with reading priors. Similarly, for *Linear Method Body*, the writing model achieves 92.66.

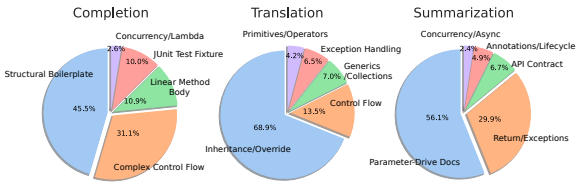

Figure 6: Distribution of semantic categories across tasks. Completion relies heavily on boilerplate, while summarization focuses on contracts.

This confirms that patterns captured during writing sessions best reflect the sequential dependencies and generative strategies critical to code completion.

Translation and Summarization. Conversely, reading artifacts (EYEMULATOR(R)) excel in comprehension-heavy tasks. On *Multi-statement Control Flow* in translation, Hybrid-Exact improves to 25.86, surpassing the writing variant (22.41). In summarization, reading priors achieve the highest overall score (42.98), reflecting the comprehension-oriented nature of distilling code into natural language.

Overall Trends. The data reveals a distinct task-dependency: generative tasks like completion benefit most from the "output-oriented" attention patterns of writing, while translation and summarization align better with the "input-processing" nature of reading fixations. While the full model remains robust, specialized priors often yield the peak performance for their respective domains.

4.4 RQ4: Ablation Studies

Table 4 (left panel) presents the ablation analysis, comparing the full EYEMULATOR model against variants with individual components removed.

Rarity Weighting. The removal of the frequency-based rarity component (w/o Frequency) causes the

Table 3: Impact of Reading (R) vs. Writing (W) priors. EYEMULATOR(Full) combines both for best results.

Group	Baseline	Ours (R)	Ours (W)	Ours (Full)
Completion (*H-Exact*)				
Structural Boilerplate	54.61	50.22	86.07	79.17
Linear Method Body	55.05	56.42	92.66	88.99
Overall	48.90	52.81	82.80	79.95
Translation (*H-Exact*)				
Multi-stmt Control	12.93	25.86	22.41	22.41
Primitives & Ops	41.67	72.22	72.22	61.11
Overall	33.95	45.55	43.53	42.72
Summarization (*BERTScore*)				
API Contract	21.45	41.11	40.96	41.35
Overall	26.37	42.98	42.83	42.97

Table 4: Ablation (left) and Attention Quality (right).

Variant	Compl.	Trans.	Attn Metric	Base	Ours
Full Model	77.96	61.02	**Summarization**		
w/o Semantic	75.62	55.92	Recency Focus ↑	0.55	1.27
w/o Frequency	56.02	55.45	Avg. Entropy ↓	88.2	60.4
w/o Monogram	71.99	56.61	**Completion**		
w/o Bi/Trigram	72.44	57.42	Confidence ↑	-0.06	0.00
Baseline	50.85	53.25	Recency Focus ↑	0.47	0.50

most significant performance drop in Code Completion, where the Hybrid-Exact score falls from 77.96 to 56.02 (-21.94 points). This sharp decline confirms that up-weighting rare n-grams is essential for generative tasks to prevent the model from collapsing into repetitive, low-entropy patterns.

Semantic Priors. In contrast, removing the Beta-derived semantic priors (w/o Semantic) primarily impacts Code Translation, reducing the H-Exact score by 5.10 points (61.02 to 55.92). This suggests that explicit knowledge of token salience (e.g., distinguishing variable declarations from delimiters) is crucial for the structural realignment required in translation tasks.

4.5 RQ5: Attention Distribution

We analyze the morphological changes in model attention using the metrics in Table 4 (right panel) and the visualizations in Figure 7.

Quantitative Shifts. EYEMULATOR induces a measurable shift toward sharper, more confident attention. In Summarization, the model reduces Average Entropy from 88.2 to 60.4 and more than doubles the Recency Focus Score (0.55 to 1.27), indicating a transition from diffuse scanning to targeted information extraction. In Completion, the Generation Confidence Score improves from -0.06 to 0.00, reflecting reduced uncertainty during token prediction.

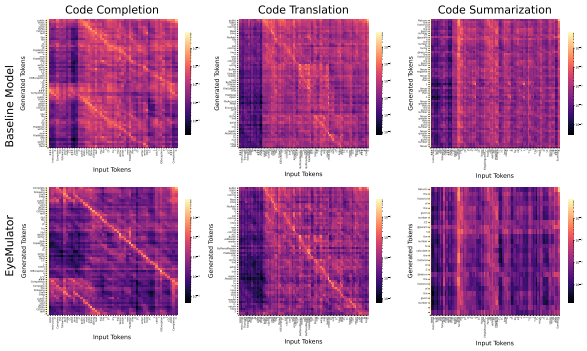

Figure 7: Attention maps: EYEMULATOR (bottom) ignores irrelevant tokens in completion/translation compared to baseline.

Qualitative Analysis. Figure 7 visualizes these shifts. In Translation (center), the diagonal alignment becomes significantly sharper compared to the baseline, showing precise token-to-token mapping. In Summarization (right), the model effectively ignores syntactic sugar to fixate on semantically dense tokens like loop conditions and return values. A broader selection of qualitative case studies, highlighting how EyeMulator mitigates specific baseline failure modes such as infinite generation loops and semantic hallucinations, is presented in Appendix C.

5 Discussion and Future Work

While EYEMULATOR demonstrates significant gains on 1B-parameter models, we identify several paths for future enhancement.

Scalability and Tasks. As a model-agnostic framework, EYEMULATOR can be applied to larger backbones (e.g., 7B or 13B parameters) via parameter-efficient fine-tuning (PEFT). We also intend to extend the methodology to tasks such as bug localization and code review, and investigate gaze-aware pre-training objectives like salience-weighted masked-token prediction.

Real-world Deployment. To evaluate practical utility, we aim to prototype real-time IDE integrations using camera-based gaze estimation to provide context-aware completions. We further plan to conduct longitudinal field studies with professional developers to assess the impact of human-aligned attention on productivity and cognitive load.

Data and Multi-modality. To move beyond laboratory-scale datasets, we are developing automated, privacy-preserving gaze collection pipelines

for open-source workflows. By augmenting eye-tracking data with behavioral signals like keystroke dynamics and navigation patterns, we aim to construct a multi-modal corpus that allows EYEMULATOR to capture the complex nuances of human program comprehension at scale.

6 Related Work

Prior work in code intelligence has advanced attention-based Transformers, yet often omits human cognitive cues. EYEMULATOR unifies these directions by integrating distilled attention priors, n-gram rarity weighting, and sequential gaze modeling into a single framework.

6.1 Human-centered AI for Software Engineering

Human-centered AI integrates cognitive insights to align models with developer workflows (Abrahão et al., 2025; Capel and Brereton, 2023; Usmani et al., 2023; Xu et al., 2023). Eye-tracking has historically illuminated how programmers manage cognitive load (Sharafi et al., 2020, 2015b; Grabinger et al., 2024; Sharafi et al., 2015a), identifying key segments to improve automated summarization (Bansal et al., 2023a; Rodeghero et al., 2014; Sharafi et al., 2015b). Recent work deepens this integration by correlating mouse interactions with neural attention (Paltenghi and Pradel, 2021), training predictive gaze models (Bansal et al., 2023b), and incorporating gaze into transformer architectures (Zhang et al., 2024) or program repair (Huber et al., 2023). Unlike these approaches, EYEMULATOR distills gaze artifacts into modular, task-agnostic priors that can be injected into any pre-trained model without architectural changes, preserving sample efficiency.

6.2 Large Language Models for Code Intelligence

LLMs such as StarCoder (Li et al., 2023), Llama-3.2 (Meta AI, 2024; Grattafiori et al., 2024), and DeepSeek-Coder (Guo et al., 2024a) have advanced code generation (Nam et al., 2024; Coignion et al., 2024; Wang et al., 2024a; Feng et al., 2020). Strategies to refine performance include Retrieval-Augmented Generation (RAG) (Wang et al., 2025; Yang et al., 2025; Guo et al., 2024b; Parvez et al., 2021), instruction tuning (Ouyang et al., 2022), and reasoning frameworks like SemCoder (Ding et al., 2024). However, models still struggle with deep semantic understanding (Nguyen et al., 2024; He et al., 2024b; Zhong and Wang, 2024; He et al., 2024a), leading to hallucinations (Liu et al., 2023; Siddiq et al., 2024; Zhang et al., 2025). Existing feedback methods often lack token-level granularity (Xu et al., 2024; Dou et al., 2024). EYEMULATOR bridges this gap by injecting gaze-derived salience directly into self-attention, enhancing semantic grounding.

6.3 Preference Learning and Model Alignment

Preference learning aligns models with developer needs beyond simple correctness (Jiang et al., 2024; Slocum et al., 2025; Wallace et al., 2024; Xiong et al., 2023). While Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022; Wang et al., 2023; Kirk et al., 2023; Wang et al., 2024b) is standard, direct optimization methods like DPO (Rafailov et al., 2024), SimPO (Meng et al., 2024), and KTO (Ethayarajh et al., 2024) offer efficient alignment. Techniques such as Group Relative Policy Optimization (GRPO) (Shao et al., 2024) further stabilize training. EYEMULATOR extends this landscape by incorporating gaze-derived salience priors as token-level feedback within DPO, enabling precise alignment with human cognitive processes.

7 Conclusion

We present EYEMULATOR, a lightweight, model-agnostic framework that injects human gaze signals into LLMs for code tasks. By mapping eye-tracking data from 27 programmers onto AST tokens, we derive semantic salience priors and n-gram gaze-transition tables, then incorporate them via a re-weighted cross-entropy loss with token-level DPO. With under 1MB of overhead and no architectural changes, EYEMULATOR delivers sizable gains in code translation, completion, and summarization. Ablation studies and attention-map visualizations confirm each component's value and show that model focus aligns with control-flow and data-dependency structures.

Data Availability

All research artifacts, including the EYEMULATOR source code, distilled gaze priors, and scripts for reproducing the experiments, are permanently archived on Zenodo at `https://zenodo.org/records/16134801`.

8 Limitations

While EYEMULATOR demonstrates significant improvements in code intelligence tasks, we acknowledge several limitations in our current study.

Model Scale and Compute. Due to computational resource constraints, our evaluation focused on models in the 1B to 7B parameter range (StarCoder-1B, Llama-3.2-1B/3B, DeepSeek-Coder-1.3B). While we hypothesize that gaze priors will scale to larger backbones (e.g., 70B+), verifying this requires further experimentation on high-performance computing clusters.

Language and Task Diversity. Our distilled priors are derived from datasets primarily consisting of Java and C#. Consequently, the efficacy of transferring these priors to structurally distinct languages (e.g., Python, Haskell) or markup languages (e.g., HTML/CSS) remains unverified. Furthermore, our study focuses on static code comprehension tasks; applying gaze signals to dynamic, interactive editing environments may require modeling temporal gaze aspects not captured here.

Participant Demographics. The gaze patterns were distilled from a cohort of 27 verified programmers. While this sample size is consistent with prior eye-tracking research, it may not fully capture the cognitive diversity of the global developer population across different experience levels, neurodiverse traits, or cultural coding practices.

9 Ethical Considerations

This work leverages human biometric data (eye-tracking) to enhance AI models. We adhered to strict ethical guidelines throughout the research process.

Potential for Misuse. We recognize the potential dual-use risk of gaze analysis technologies in workplace surveillance. We explicitly condemn the use of such methods for monitoring employee productivity or engagement. EYEMULATOR is designed solely to extract generalized cognitive patterns to improve AI tooling, not to assess or track individual developers. We urge the community to maintain strict boundaries between aggregate model training and individual user monitoring.

Data Privacy and Dataset Usage. This study utilizes the publicly available *EyeTrans* dataset, adhering to the ethical protocols and informed consent procedures established by its creators. The data is strictly anonymized with all PII removed. Our work further ensures privacy by distilling only aggregated statistical distributions, preventing the reconstruction of individual user behaviors.

References

Silvia Abrahão, John Grundy, Mauro Pezzè, Margaret-Anne Storey, and Damian A. Tamburri. 2025. Software engineering by and for humans in an AI era. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46.

Aakash Bansal, Bonita Sharif, and Collin McMillan. 2023a. Towards modeling human attention from eye movements for neural source code summarization. *Proceedings of the ACM on Human-Computer Interaction*, 7(ETRA):1–19.

Aakash Bansal, Chia-Yi Su, Zachary Karas, Yifan Zhang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2023b. Modeling programmer attention as scanpath prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1732–1736. IEEE.

Tara Capel and Margot Brereton. 2023. What is human-centered about human-centered ai? a map of the research landscape. In *Proceedings of the 2023 CHI conference on human factors in computing systems*, pages 1–23.

Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, pages 79–89.

Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Preprint*, arXiv:2406.01006.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. StepCoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint*.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. KTO: Model alignment as prospect theoretic optimization. *arXiv preprint*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Lisa Grabinger, Florian Hauser, Christian Wolff, and Jürgen Mottok. 2024. On eye tracking in software engineering. *SN Computer Science*, 5(6):729.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Kadian, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. DeepSeek-coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint*.

Yucan Guo, Zixuan Li, Xiaolong Jin, Yantao Liu, Yutao Zeng, Wenxuan Liu, Xiang Li, Pan Yang, Long Bai, Jiafeng Guo, and 1 others. 2024b. Retrieval-augmented code generation for universal information extraction. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 30–42. Springer.

E. Harth and P. Dugerdil. 2017. Program understanding models: An historical overview and a classification. In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT)*, volume 1, pages 402–413. SciTePress.

Fusen He, Juan Zhai, and Minxue Pan. 2024a. Beyond code generation: Assessing code llm maturity with postconditions. *arXiv preprint arXiv:2407.14118*.

Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. 2024b. Exploring demonstration retrievers in rag for coding tasks: Yeas and nays! *arXiv preprint arXiv:2410.09662*.

Y. Huang, K. Leach, Z. Sharafi, T. Santander, and W. Weimer. 2020. Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 456–468. ACM.

Dominik Huber, Matteo Paltenghi, and Michael Pradel. 2023. Where to look when repairing code? Comparing the attention of neural models and developers. *arXiv preprint*.

Ruili Jiang, Kehai Chen, Xuefeng Bai, Zhixuan He, Juntao Li, Muyun Yang, Tiejun Zhao, Liqiang Nie, and Min Zhang. 2024. A survey on human preference learning for large language models. *arXiv preprint*.

Robert Kirk, Ishita Mediratta, Christoforos Nalmpantis, Jelena Luketina, Eric Hambro, Edward Grefenstette, and Roberta Raileanu. 2023. Understanding the effects of rlhf on llm generalisation and diversity. *arXiv preprint arXiv:2310.06452*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Akiki, and 1 others. 2023. StarCoder: May the source be with you! *arXiv preprint*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *arXiv preprint*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Yu Meng, Mengzhou Xia, and Danqi Chen. 2024. SimPO: Simple preference optimization with a reference-free reward. *arXiv preprint*.

Meta AI. 2024. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. Meta AI Blog. `https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/`.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Thu-Trang Nguyen, Thanh Trong Vu, Hieu Dinh Vo, and Son Nguyen. 2024. An empirical study on capability of large language models in understanding code semantics. *arXiv preprint*.

M. P. O'Brien. 2003. Software comprehension: A review and research direction. Technical report, Department of Computer Science & Information Systems, University of Limerick.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *arXiv preprint*.

Matteo Paltenghi and Michael Pradel. 2021. Thinking like a developer? Comparing the attention of humans with neural models of code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 867–879, Melbourne, Australia. IEEE.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint*.

Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving

automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 390–401, New York, NY, USA. Association for Computing Machinery.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. 2015a. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 96–103.

Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering*, 25(5):3128–3174.

Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015b. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, 67:79–107.

Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. 2024. Quality assessment of ChatGPT generated code and their use by developers. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 152–156, Lisbon Portugal. ACM.

Stewart Slocum, Asher Parker-Sartori, and Dylan Hadfield-Menell. 2025. Diverse preference learning for capabilities and alignment. In *The Thirteenth International Conference on Learning Representations*.

Usman Ahmad Usmani, Ari Happonen, and Junzo Watada. 2023. Human-centered artificial intelligence: Designing for user empowerment and ethical considerations. In *2023 5th international congress on human-computer interaction, optimization and robotic applications (HORA)*, pages 1–7. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Bram Wallace, Meihua Dang, Rafael Rafailov, Linqi Zhou, Aaron Lou, Senthil Purushwalkam, Stefano Ermon, Caiming Xiong, Shafiq Joty, and Nikhil Naik. 2024. Diffusion model alignment using direct preference optimization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8228–8238.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Yuanhao Wang, Qinghua Liu, and Chi Jin. 2023. Is rlhf more difficult than standard rl? a theoretical perspective. *Advances in Neural Information Processing Systems*, 36:76006–76032.

Zhichao Wang, Bin Bi, Shiva Kumar Pentyala, Kiran Ramnath, Sougata Chaudhuri, Shubham Mehrotra, Xiang-Bo Mao, Sitaram Asur, and 1 others. 2024b. A comprehensive survey of llm alignment techniques: Rlhf, rlaif, ppo, dpo and more. *arXiv preprint arXiv:2407.16216*.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. Coderag-bench: Can retrieval augment code generation? *Preprint*, arXiv:2406.14497.

Wei Xiong, Hanze Dong, Chenlu Ye, Ziqi Wang, Han Zhong, Heng Ji, Nan Jiang, and Tong Zhang. 2023. Iterative preference learning from human feedback: Bridging theory and practice for rlhf under kl-constraint. *arXiv preprint arXiv:2312.11456*.

Wei Xu, Marvin J Dainoff, Liezhong Ge, and Zaifeng Gao. 2023. Transitioning to human interaction with ai systems: New challenges and opportunities for hci professionals to enable human-centered ai. *International Journal of Human–Computer Interaction*, 39(3):494–518.

Wenda Xu, Daniel Deutsch, Mara Finkelstein, Juraj Juraska, Biao Zhang, Zhongtao Liu, William Yang Wang, Lei Li, and Markus Freitag. 2024. LLMRefine: Pinpointing and refining large language models via fine-grained actionable feedback. *arXiv preprint*.

Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*.

Yifan Zhang, Jiliang Li, Zachary Karas, Aakash Bansal, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Eyetrans: Merging human and machine attention for neural code summarization. *Proceedings of the ACM on Software Engineering*, 1(FSE):115–136.

Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2025. LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *arXiv preprint*.

Li Zhong and Zilong Wang. 2024. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 21841–21849.

A Evaluation Metrics

To ensure reproducibility, we provide formal definitions for all metrics used to evaluate task performance (RQ2–RQ4) and attention alignment (RQ5).

A.1 Task Performance Metrics

A.1.1 Code Translation and Completion

For Java-to-C# translation and code completion, we employ three metrics to capture both lexical and semantic correctness:

- **Hybrid Exact Match (H-Exact):** To account for minor formatting variations while rewarding precision, we calculate a weighted average of strict exact match and substring inclusion:

$$\text{H-Exact} = 0.5 \times \mathbb{I}(y = \hat{y}) + 0.5 \times \mathbb{I}(\hat{y} \in y) \tag{1}$$

  where y is the ground truth, $\hat{y}$ is the generated code, and $\mathbb{I}(\cdot)$ is the indicator function.

- **CodeBLEU:** Unlike standard BLEU, CodeBLEU integrates syntactic and semantic properties. It is computed as the weighted sum of four components:

$$\begin{aligned}\text{CodeBLEU} = w_1\text{BLEU} + w_2\text{BLEU}_{\text{weighted}} \\ + w_3\text{Match}_{\text{ast}} + w_4\text{Match}_{\text{df}}\end{aligned} \tag{2}$$

  where $\text{BLEU}_{\text{weighted}}$ gives higher weight to keywords, $\text{Match}_{\text{ast}}$ measures Abstract Syntax Tree similarity, and Match_{df} measures dataflow graph similarity.

- **CrystalBLEU:** A variant of BLEU optimized for code that filters out "trivially shared" n-grams (e.g., frequent syntax like `public void`) to prevent inflated scores. It calculates n-gram precision only on a distinct set of n-grams not appearing in the top-k most frequent occurrences in the training corpus.

A.1.2 Code Summarization

For Java-to-Natural Language summarization, we use standard text-generation metrics:

- **ROUGE-L:** Measures the Longest Common Subsequence (LCS) between the candidate summary and the reference, capturing sentence-level structure.

- **METEOR:** Computes the harmonic mean of precision and recall, incorporating stemming and synonym matching to capture semantic overlap.

- **BERTScore:** Computes the similarity between candidate and reference summaries using contextual embeddings from a pre-trained BERT model:

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{y_j \in y} \mathbf{x}_i^\top \mathbf{y}_j \tag{3}$$

  where $\mathbf{x}_i$ and $\mathbf{y}_j$ are the embedding vectors for tokens in the candidate and reference, respectively.

A.2 Attention Alignment Metrics (RQ5)

To quantify how well EyeMulator aligns model focus with relevant code regions (RQ5), we analyze the final-layer attention weights $a = (a_1, \ldots, a_n)$ over the input sequence of length n.

- **Generation Confidence Score (GCS):** Measures the model's certainty during decoding. Higher values indicate the model assigns higher probability to the selected tokens.

$$\text{GCS} = \frac{1}{T} \sum_{t=1}^{T} \log P(y_t \mid y_{<t}, x) \tag{4}$$

  where T is the length of the generated sequence y.

- **Recency Focus Score (RFS):** Quantifies the proportion of attention mass allocated to the most recent context window (last k tokens, where $k = 20$), often critical for code completion tasks.

$$\text{RFS} = \frac{\sum_{i=n-k+1}^{n} a_i}{\sum_{i=1}^{n} a_i} \tag{5}$$

- **Average Focus Score (AFS):** Measures the intensity of attention specifically on *semantically critical* tokens (as identified by the eye-tracking ground truth). Let C be the set of indices corresponding to critical tokens:

$$\text{AFS} = \frac{1}{|C|} \sum_{i \in C} a_i \tag{6}$$

- **Attention Entropy:** Measures the dispersion of the attention distribution. Lower entropy indicates sharper, more confident focus; higher entropy suggests diffuse attention.

$$\text{Entropy} = -\sum_{i=1}^{n} a_i \log a_i \tag{7}$$

B Implementation Details

To ensure the reproducibility of our results, we provide the comprehensive configuration for the EyeMulator and baseline models.

B.1 Model and Loss Configuration

We utilized the `Llama-3.2-1B` architecture for all experiments. While the baseline uses the standard Causal Language Modeling (CLM) objective, EyeMulator employs a custom loss function that integrates human attention weights. Based on our training implementation, we perform **full fine-tuning** of all parameters.

B.2 Hyperparameters

We maintained consistent hyperparameters across all tasks. These are summarized in Table 5.

Table 5: Experimental Hyperparameters

Hyperparameter	Value
Base Model	Llama-3.2-1B
Optimizer	AdamW
Learning Rate	2×10^{-5}
Effective Batch Size	16
Max Sequence Length	1024 tokens
Training Epochs	3
Weight Decay	0.01
Precision	FP32 (Full Precision)
Hardware	1× NVIDIA L40S

B.3 Attention Prior Processing

For the "Writing-Mode" variants, we mapped semantic labels (e.g., `MethodDeclaration`) to token indices. The attention weights were injected into the loss function via a lookup table, where the loss for token i is scaled by its corresponding human fixation weight w_i.

C Qualitative Case Studies

We present specific cases where EyeMulator demonstrates superior logic and conciseness compared to the baseline model.

C.1 Case Study 1: Java-to-C# Translation

In this example, the baseline model fails to correctly terminate the generation, resulting in a syntax error and unreachable code. EyeMulator identifies the high-level semantic intent and utilizes a built-in C# method.

Input (Java Code)

```
public int FindFirstRecordLocBySid(short sid) {
    int index = 0;
   for (IEnumerator iterator = records.GetEnumerator();
         iterator.MoveNext(); ) {
        Record record = (Record)iterator.Current;
        if (record.Sid == sid) { return index; }
        index++;
    }
    return -1;
}
```

Baseline (Llama-3.2-1B Baseline)

```
public int FindFirstRecordLocBySid(int sid) {
    int index = 0;
    foreach (Record record in records.Records) {
        if (record.Sid == sid) return index;
    }
    return -1;
} return -2;} // Hallucinated syntax
```

EyeMulator (Ours)

```
public int FindFirstRecordLocBySid(short sid) {
    // Successfully identified idiomatic IndexOf
    int index = records.IndexOf(sid);
    return index;
}
```

Figure 8: Translation example: EyeMulator avoids the "unending generation" failure seen in the baseline.

C.2 Case Study 2: Code Summarization

This case illustrates the "repetition loop" common in smaller LLMs. While the baseline generates redundant Javadoc-style constraints, EyeMulator provides a focused summary of the method's behavior.

Input (Java Method)

```
public String getSqlWithValues() {
    if( namedParameterValues.size() == 0 ) {
      return super.getSqlWithValues();
    }
    return result.toString();
}
```

Baseline Output *Returns a SQL query with the named parameter values added. @return the SQL query. Never null. Always returns non empty. Never returns an empty string. Never returns a non-empty string... [Truncated]*

EyeMulator Output *Returns the SQL query with the values of the parameters set.*

Figure 9: Summarization example: EyeMulator maintains semantic focus and avoids repetition.