EYEMULATOR: Improving Code Language Models by Mimicking Human Visual Attention

Yifan Zhang[1] **Chen Huang**[2] **Yueke Zhang**[1] **Jiahao Zhang**[1]
Toby Jia-Jun Li[3] **Collin McMillan**[3] **Kevin Leach**[1] **Yu Huang**[1]

Vanderbilt University[1] National University of Singapore[2] University of Notre Dame[3]
{yifan.zhang.2,yueke.zhang,jiahao.zhang,kevin.leach,yu.huang}@vanderbilt.edu
huang_chen@nus.edu.sg {toby.j.li,cmc}@nd.edu

Abstract

Code Language Models (CodeLLMs) learn token importance from data correlations, whereas human developers attend selectively to semantically salient code. We present EYEMULATOR, a model-agnostic method that injects human visual-attention priors into CodeLLM fine-tuning without architectural changes. EYEMULATOR distills eye-tracking data into semantic salience and gaze-transition priors, then uses them to reweight token-level training losses. Across six backbones, two data regimes, and three CodeXGLUE tasks, the reported configurations yield positive matched-metric deltas in all 36 model-task-setting cells. Effects are largest for structure-preserving completion and translation, while summarization shows smaller but positive METEOR deltas. Session-mode and component-ablation analyses further show that reading, writing, semantic, and transition-derived priors provide complementary signal. Human-attention artifacts are available at https://zenodo.org/records/17205682.

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 CodeLLMs to learn human cognitive patterns di-

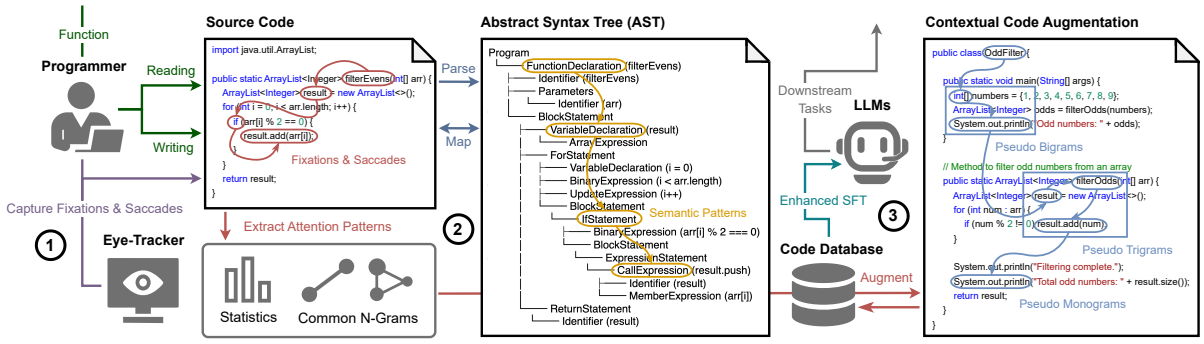

Figure 1: Overview of EYEMULATOR. Human gaze traces are aligned to AST tokens, distilled into semantic salience and transition priors, and converted into pseudo-scan paths that guide weighted SFT and token-level preference optimization without changing the base CodeLLM architecture.

rectly 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 CodeXGlue tasks: code completion, Java-to-C# translation, and Java-to-English summarization. To test robustness, we run six small CodeLLM backbones across low-data and full-data regimes, with seed 42 and seed 12345 used for robustness checks. The cleaned results show positive quality-safe gains in every model–task–setting cell. Completion and translation provide the strongest evidence, with consistent Exact Match improvements across all six models and both data regimes; summarization is more recipe-sensitive but remains positive under METEOR after targeted tuning and DeepSeek tokenizer correction. Session-mode, component-ablation, and attention diagnostics further support the mechanism behind these gains.

In summary, this paper contributes: (1) a portable pipeline for converting raw human gaze traces into reusable semantic salience and sequential transition priors over code tokens; (2) a model-agnostic fine-tuning objective that incorporates these priors through token-level weighting and preference alignment without requiring architectural changes; and (3) a six-backbone evaluation across multiple CodeXGlue tasks and data regimes, showing that gaze-derived signals provide consistent quality-safe gains while revealing when human-attention cues are robust, task-sensitive, and complementary.

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.

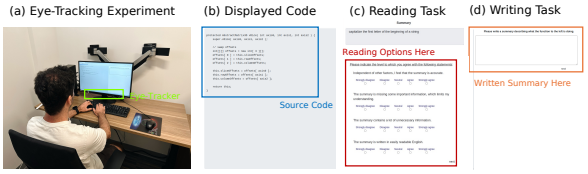

Figure 2: EyeTrans gaze data used to derive human-attention priors. Fixations over Java snippets are aligned with AST tokens and separated into reading and writing sessions before statistical distillation.

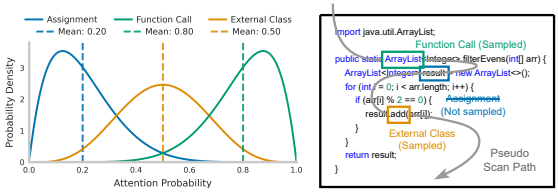

Figure 3: Pseudo-scan-path generation. Semantic salience priors identify important code tokens, while transition statistics connect them into gaze-like token paths for training supervision.

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: $w_j = w_{\text{base}} + (\log(\text{freq}(g_j) + 2))^{-1} + \mathbb{E}[\theta_{s_j}]$.

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

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

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 ($\mathcal{L}_{\text{DPO}}$).

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 five research questions: (RQ1) How faithfully do our distilled artifacts capture human attention patterns? (RQ2) Does mimicking human attention improve performance on downstream code-intelligence tasks? (RQ3) How do reading-derived versus writing-derived priors differ in their impact across tasks? (RQ4) How do the individual components of EYEMULATOR contribute to overall gains? (RQ5) Does the fine-tuned 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 CodeXGlue (Lu et al., 2021), selecting three tasks to test generalization. For *Code Translation* (Java to C#), we use the standard Java-to-C# split. For *Code Summarization* (Java to English), we use 16,490 training, 518 validation, and 1,095 test examples. For *Code Completion*, we sample 20% of the CodeXGlue Java completion corpus, yielding 1,633 training, 1,005 validation, and 1,318 test examples. We report two data regimes: a low-data setting with 200 training examples and a full-data setting using the available training split. Completion and translation are evaluated with Exact Match, while summarization is evaluated with METEOR. Appendix A provides the metric definitions.

Models and Baselines. To demonstrate model agnosticism, we apply EYEMULATOR to six backbones: StarCoderBase-1B, Llama-3.2-1B, DeepSeek-Coder-1.3B, Qwen2.5-Coder-1.5B, SmolLM3-3B, and Granite-Code-2B. Each baseline is fine-tuned on the same task data without gaze weights. EYEMULATOR uses task-level LoRA/QLoRA recipes, with code completion and translation using a rank-32, weight-4 recipe and summarization using lower-rank recipes selected from completed quality-safe variants. For robustness, we run the clean recipe on seeds 42 and 12345 and report best quality-safe results only when generated outputs pass qualitative sanity checks.

Implementation Details. All experiments use HuggingFace `transformers` with parameter-efficient LoRA/QLoRA adapters. Input sequences are processed with a maximum length of 1024 tokens. Completion and translation use rank-32 LoRA with gaze weight 4 as the clean recipe; summarization uses lower-rank recipes selected from quality-safe sweeps. We use AdamW-style optimization, task-specific learning rates, and seeds 42 and 12345 for robustness checks. Detailed training configurations and attention-prior alignment steps are documented in Appendix C.

4 Result Analysis

We organize the results from broadest to most diagnostic. We first establish the main effect across backbones, tasks, and data regimes (RQ2), then use session-mode and component-ablation analyses to explain which gaze signals matter (RQ3–RQ4). Finally, we retain the attention-map analysis as an appendix diagnostic rather than as the primary evi-

Table 1: Representative gaze patterns extracted from token-aligned fixations. Frequent transitions show that programmers repeatedly move between declarations, parameters, and control-flow tokens rather than scanning code uniformly.

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

dence (RQ5).

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.

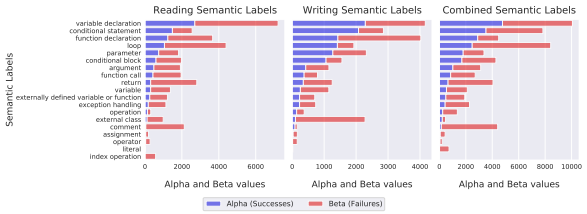

Figure 4: Estimated Beta parameters for semantic token classes. Larger α_s values indicate token types that consistently attract developer fixations, with variable declarations emerging as a primary attention target.

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.

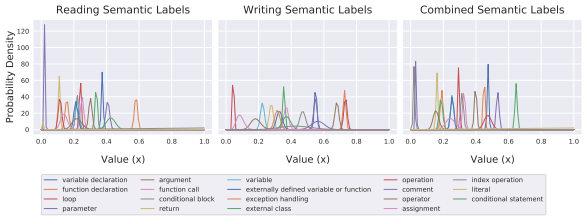

Figure 5: Smoothed Beta density functions for representative semantic classes. Narrow peaks indicate consistently focused attention, while broader curves capture context-dependent inspection of control-flow constructs.

4.2 RQ2: Main Cross-Task Results

RQ2 asks whether gaze-derived priors improve downstream CodeLLM fine-tuning across tasks, data regimes, and backbones. The main result is that EYEMULATOR improves every evaluated cell under the matched task metric: 36 positive gains across six backbones, three tasks, and two data settings. Table 2 reports the selected quality-safe run for each cell, and Figure 6 summarizes the same evidence as a model-by-task profile.

Structure-Preserving Code Tasks. The strongest evidence comes from completion and translation, where the target output preserves executable code structure. Completion gains

Table 2: Main cross-task evaluation across six backbones, two data regimes, and three CodeXGlue tasks. Gray rows report EYEMULATOR scores; parentheses show relative improvement over the matched baseline under the task metric.

Method	Low-data setting			Full-data setting		
	Completion Exact	Translation Exact	Summarization METEOR	Completion Exact	Translation Exact	Summarization METEOR
Llama-3.2-1B Baseline	75.42	38.28	25.83	81.94	56.94	28.91
Llama-3.2-1B EYEMULATOR	81.49 (+8.0%)	41.99 (+9.7%)	27.58 (+6.8%)	85.13 (+3.9%)	60.53 (+6.3%)	31.59 (+9.3%)
DeepSeek-Coder-1.3B Baseline	67.91	44.50	30.23	74.43	66.87	30.79
DeepSeek-Coder-1.3B EYEMULATOR	78.60 (+15.8%)	49.28 (+10.8%)	30.65 (+1.4%)	99.32 (+33.4%)	69.98 (+4.7%)	32.40 (+5.2%)
StarCoderBase-1B Baseline	73.67	37.44	30.53	74.13	58.25	31.75
StarCoderBase-1B EYEMULATOR	98.94 (+34.3%)	42.46 (+13.4%)	31.24 (+2.3%)	99.62 (+34.4%)	61.36 (+5.3%)	33.95 (+6.9%)
Qwen2.5-Coder-1.5B Baseline	77.85	36.36	18.41	79.89	57.18	28.88
Qwen2.5-Coder-1.5B EYEMULATOR	96.81 (+24.4%)	42.94 (+18.1%)	20.71 (+12.5%)	99.70 (+24.8%)	62.08 (+8.6%)	30.86 (+6.9%)
SmolLM3-3B Baseline	77.69	37.56	15.29	81.87	55.74	27.12
SmolLM3-3B EYEMULATOR	92.64 (+19.2%)	41.39 (+10.2%)	20.23 (+32.3%)	99.54 (+21.6%)	60.17 (+7.9%)	28.70 (+5.8%)
Granite-Code-2B Baseline	71.93	37.68	23.81	73.44	55.50	32.20
Granite-Code-2B EYEMULATOR	74.66 (+3.8%)	40.67 (+7.9%)	27.39 (+15.0%)	77.31 (+5.3%)	61.48 (+10.8%)	33.07 (+2.7%)

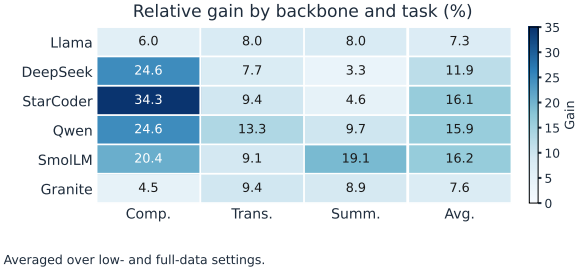

Figure 6: Backbone-by-task profile of EYEMULATOR gains. Each cell averages relative improvement over low- and full-data settings; darker colors and annotations indicate larger improvements.

are largest for code-specialized backbones after targeted recipe selection, reaching +34.4% relative improvement while remaining positive for all other models. Translation is less peaked but more uniformly stable, with relative gains up to +18.1% in the low-data setting and +10.8% in the full-data setting. These patterns suggest that gaze-derived priors are especially useful when semantic attention can directly constrain valid code tokens.

Natural-Language Summarization. Summarization is more sensitive to decoding, tokenizer behavior, and LoRA recipe choice than the two code-structure tasks, because the target output no longer preserves the input program structure token by token. After excluding pathological outputs and applying the DeepSeek tokenizer fix, EYEMULATOR improves every summarization cell under METEOR, suggesting that gaze priors still provide useful semantic guidance even when code understanding must be expressed in natural language.

Backbone-Level View. Figure 6 shows why evaluating multiple backbones is more informative than reporting a larger table alone. The average relative gain is similar for StarCoderBase, Qwen2.5-Coder, and SmolLM3 (about 16%), while Llama-3.2 and Granite-Code show smaller but still consistent gains (about 7–8%). This spread supports a model-agnostic interpretation without hiding backbone sensitivity: code-specialized backbones benefit most on completion, whereas Granite and Llama indicate that the signal is not limited to the strongest completion-tuned models.

Model and Data-Regime Robustness. Across Table 2, the selected EYEMULATOR runs improve every task, backbone, and data regime under the matched metric. The results support the main claim under a conservative interpretation: human-attention priors provide consistent positive signal across architectures and data regimes, with the strongest effect on structure-preserving code generation and a smaller but positive effect on natural-language summarization. The breadth of this pattern is important because it rules out a single-model or single-task explanation. It also motivates the next two analyses, which ask which session source and which gaze-derived components account for the observed gains.

Taken together, RQ2 establishes the outcome-level effect. RQ3 and RQ4 then unpack this effect mechanistically: RQ3 varies the source of the human-attention prior, while RQ4 removes individual signal components from the full recipe.

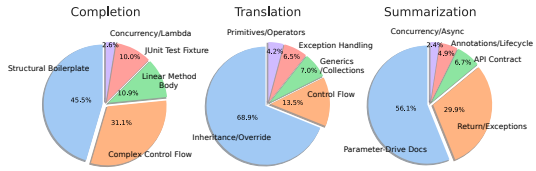

Figure 7: Semantic-category profiles across tasks. Completion emphasizes structural boilerplate, translation centers on inheritance/override patterns, and summarization shifts toward API and return semantics.

Table 3: Session-mode results on current Llama-3.2-1B full-data runs. Gray cells mark the best variant per subcategory; parentheses show relative improvement over baseline.

Group	Base	Read	Write	Full
Completion (*Exact*)				
Concurrency/Lambda	64.46	67.47 (+4.7%)	70.48 (+9.3%)	72.89 (+13.1%)
Control Flow	74.61	77.95 (+4.5%)	79.51 (+6.6%)	79.51 (+6.6%)
Boilerplate	98.89	99.12 (+0.2%)	99.12 (+0.2%)	99.12 (+0.2%)
Linear Body	100.00	100.00 (+0.0%)	100.00 (+0.0%)	100.00 (+0.0%)
JUnit Fixture	72.60	74.43 (+2.5%)	74.43 (+2.5%)	74.89 (+3.1%)
Overall	81.94	83.84 (+2.3%)	84.75 (+3.4%)	85.13 (+3.9%)
Translation (*Exact*)				
Multi-stmt Ctrl.	13.93	22.13 (+58.8%)	20.49 (+47.1%)	21.31 (+52.9%)
Exceptions	25.00	31.25 (+25.0%)	30.00 (+20.0%)	33.75 (+35.0%)
Generics/Coll.	42.42	45.45 (+7.1%)	48.48 (+14.3%)	45.45 (+7.1%)
Inheritance	25.00	31.25 (+25.0%)	31.25 (+25.0%)	31.25 (+25.0%)
Primitives/Ops	72.99	74.19 (+1.6%)	73.50 (+0.7%)	72.31 (-0.9%)
Overall	57.66	60.53 (+5.0%)	59.81 (+3.7%)	59.33 (+2.9%)
Summarization (*METEOR*)				
API Contract	25.78	27.78 (+7.8%)	28.01 (+8.6%)	27.81 (+7.9%)
Async/Concur.	25.85	28.79 (+11.3%)	26.87 (+3.9%)	26.30 (+1.7%)
Annotations	29.72	31.52 (+6.1%)	31.07 (+4.6%)	29.58 (-0.5%)
Param Docs	29.84	31.88 (+6.8%)	32.55 (+9.1%)	32.61 (+9.3%)
Return/Except.	30.88	34.53 (+11.8%)	34.17 (+10.6%)	34.38 (+11.3%)
Overall	28.91	31.43 (+8.7%)	31.22 (+8.0%)	30.88 (+6.8%)

4.3 RQ3: Session-Mode Analysis

Figure 7 shows that semantic categories vary across tasks, motivating a comparison of reading-derived (EyeMulator(R)) and writing-derived (EyeMulator(W)) priors. We rerun this analysis on current Llama-3.2-1B full-data artifacts, using Exact Match for completion and translation and METEOR for summarization. Table 3 reports all baseline and prior variants by group.

Completion subcategories. Writing-derived and full priors are strongest for code-production cases. The full variant leads on Concurrency/Lambda (+13.1%), JUnit Test Fixture (+3.1%), and completion (+3.9%), while writing ties on Control Flow and Boilerplate. This suggests that writing traces capture generation-oriented dependencies, with the cleaned runs giving a more conservative estimate than earlier validation results.

Translation and summarization subcategories. Reading-derived priors are strongest for translation overall (+5.0%) and for Multi-statement Control Flow, Inheritance, and Primitives & Operators. The full variant is strongest for Exceptions, and writing is strongest for Generics/Collections. Summarization is more mixed under METEOR: reading leads Async/Concurrency, Annotations, Return/Exception Semantics, and the overall aggregate, while writing leads API Contract and the full model leads Parameter Docs. These results indicate that comprehension-oriented reading priors help many input-understanding cases, but no single session mode dominates all natural-language summary categories.

Combined session effect. The full-prior variant remains robust, especially for completion, but the current Llama results show a more nuanced pattern than the previous draft: specialized session priors often win individual rows. We therefore treat session mode as a task- and category-dependent design choice rather than as a universally monotonic improvement from combining reading and writing statistics.

4.4 RQ4: Component Ablation

RQ4 examines whether the full gaze signal matters beyond any single component. Table 4 reports complete component-removal sweeps spanning all three tasks: completion on Llama-3.2-1B, and translation and summarization on DeepSeek-Coder and StarCoder. Removing either AST salience or rarity/transition flow reduces performance relative to the full recipe, and randomizing the weights also underperforms Full. This pattern supports the full EyeMulator recipe: semantic salience and sequential gaze flow are complementary rather than interchangeable.

4.5 RQ5: Attention Distribution

We treat attention distribution as diagnostic evidence rather than as part of the current six-backbone confirmation grid. Table 5 recomputes the earlier attention-quality view on the selected current Llama-3.2-1B full-data adapters, using 24 held-out examples per task. The clearest current signal is improved generation confidence, especially for summarization; distributional attention measures are smaller and task-dependent, so we use them to interpret behavior rather than to replace the main quality results in Table 2.

Table 4: Component ablation across complete variant sweeps. Completion and translation use Exact Match and summarization uses METEOR; drops are relative to the Full EYEMULATOR recipe.

Backbone / Task	Variant	Score	Drop
Llama Completion	Full	85.13	–
	No semantic	83.69	-1.7%
	No rare/flow	83.99	-1.3%
	Random	84.37	-0.9%
DeepSeek Translation	Full	65.66	–
	No semantic	53.13	-19.1%
	No rare/flow	53.13	-19.1%
	Random	52.44	-20.1%
DeepSeek Summ.	Full	33.92	–
	No semantic	30.11	-11.2%
	No rare/flow	30.22	-10.9%
	Random	30.22	-10.9%
StarCoder Translation	Full	64.97	–
	No semantic	50.35	-22.5%
	No rare/flow	51.04	-21.4%
	Random	51.74	-20.4%
StarCoder Summ.	Full	33.41	–
	No semantic	30.92	-7.5%
	No rare/flow	30.97	-7.3%
	Random	31.09	-7.0%

Qualitative Analysis. Figure 8 visualizes final-layer attention from generated tokens back to input tokens. We keep this qualitative view as an illustrative diagnostic rather than as final quantitative evidence. A broader selection of qualitative case studies is presented in Appendix D.

5 Discussion

EYEMULATOR builds on a simple observation: developers do not process code uniformly. They revisit declarations, control-flow predicates, API contracts, and other semantic anchors while skimming boilerplate. Our results suggest that compact CodeLLMs benefit when this selectivity is distilled into training-time priors rather than architecture changes.

Why the gains are task-dependent. The largest improvements appear in completion and translation, where outputs preserve program structure and exact token choices matter. In these settings, gaze-derived weights help the model prioritize variables, signatures, and control-flow anchors that constrain valid code. Summarization is more diffuse because the output is natural language, making gains more dependent on recipe selection and tokenizer behavior. This helps explain why summarization gains are smaller yet still positive under METEOR.

Table 5: Attention diagnostics on current Llama-3.2-1B full-data adapters. Arrows indicate the expected direction for each metric; gray cells mark the better score, and the table is used as diagnostic rather than primary quality evidence.

Task	Metric	Baseline	EYEMULATOR	p-value
Completion	Generation Confidence ↑	-0.0168	-0.0123	3.78e-01
	Recency Focus ↑	0.06942	0.06941	8.26e-04
	Avg. Focus ↑	0.80347	0.80345	2.44e-01
	Entropy ↓	8.62056	8.62055	2.18e-01
Translation	Generation Confidence ↑	-0.0578	-0.0407	1.80e-01
	Recency Focus ↑	0.35290	0.35314	9.12e-03
	Avg. Focus ↑	0.76240	0.76273	4.48e-03
	Entropy ↓	5.94433	5.94471	5.81e-03
Summarization	Generation Confidence ↑	-0.9126	-0.6794	2.68e-03
	Recency Focus ↑	0.18572	0.18563	2.80e-03
	Avg. Focus ↑	0.73555	0.73555	9.16e-01
	Entropy ↓	7.11796	7.11802	2.13e-01

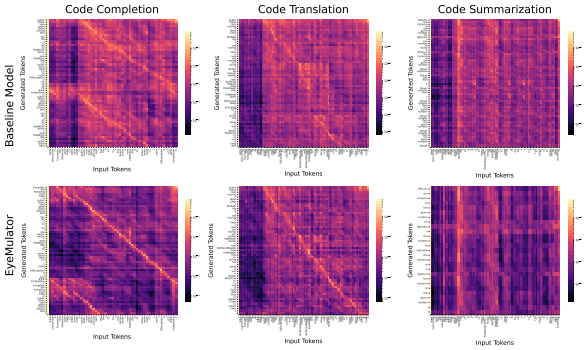

Figure 8: Qualitative attention diagnostic. Compared with the baseline (top), EYEMULATOR (bottom) concentrates more mass on semantically relevant input tokens, illustrating the intended effect of gaze-derived training priors.

Why modular priors matter. Because EYEMULATOR changes only the training objective and data weights, it can be applied across heterogeneous backbones without requiring a custom transformer. The component ablation further shows that semantic salience and transition/rarity flow are complementary: removing either signal consistently weakens the full recipe. This supports the view that human attention is useful not as a single heuristic, but as a structured prior combining what developers inspect and how they move through code.

Interpreting diagnostic evidence. The attention diagnostics serve as behavioral support, not the primary evaluation target. They show that gaze-informed fine-tuning can shift model behavior in interpretable ways, while downstream task quality remains the central evidence. Thus, EYEMULATOR's main claim rests on improvements across the evaluation grid, with attention analyses explaining why those gains are plausible.

6 Related Work

Prior work in code intelligence has advanced attention-based Transformers, yet often omits human cognitive cues about how developers actually inspect code. EYEMULATOR unifies these directions by integrating distilled attention priors, n-gram rarity weighting, and sequential gaze modeling into a single framework that can guide standard CodeLLM fine-tuning.

6.1 Human-centered AI for Software Engineering

Human-centered AI integrates cognitive insights to align models with developer workflows (Abrahão et al., 2025; Zhang et al., 2025a; Karas et al., 2024; Li et al., 2024). 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; Zhang et al., 2025b; Feng et al., 2020). Strategies to refine performance include Retrieval-Augmented Generation (RAG) (Wang et al., 2025; Yang et al., 2025b; 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., 2024; Zhong and Wang, 2024; Yang et al., 2025a), leading to hallucinations (Liu et al., 2023; Siddiq et al., 2024; Zhang et al., 2025c). 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; Fang et al., 2025; Xiong et al., 2023). While Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022; Zhang and Leach, 2025; Kirk et al., 2023; Wang et al., 2024) 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 CodeLLM fine-tuning. By mapping eye-tracking data from 27 programmers onto AST tokens, we derive semantic salience and gaze-transition priors, then incorporate them into parameter-efficient fine-tuning. Across six backbones, three tasks, and two data regimes, EYEMULATOR yields positive quality-safe gains in every evaluated cell, with the strongest effects on completion and translation and smaller but consistent gains on summarization. Session-mode and component-ablation analyses, together with appendix diagnostics, show that gaze-derived signals change model behavior in interpretable, task-dependent ways.

Acknowledgments

We thank the study participants from Vanderbilt University and the University of Notre Dame for contributing to the human gaze data that supports this work. We also thank our collaborators and lab members for their feedback on the eye-tracking setup, artifact preparation, and manuscript presentation.

8 Limitations

While EYEMULATOR demonstrates significant improvements in code intelligence tasks, we acknowledge several limitations in our current study, spanning model scale, language coverage, and participant demographics.

Model Scale and Compute. Due to computational resource constraints, our evaluation focuses on small code language models from roughly 1B to 3B parameters. This range allows broad coverage across six backbones and two data regimes, but does not establish scaling behavior for 7B, 13B, or larger systems.

Language and Task Diversity. Our priors are distilled from gaze recordings on **Java** code (EyeTrans) and evaluated on Java/C# static code comprehension tasks, so transfer to structurally distinct languages (e.g., Python, Haskell) or markup (HTML/CSS), as well as to dynamic, interactive editing environments that require temporal gaze modeling, remains unverified.

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

Potential for Misuse. Gaze analysis carries a dual-use risk in workplace surveillance. EYEMULATOR is designed only to distill *aggregated* cognitive patterns for model training, not to assess or track individual developers, and we discourage any use that blurs this boundary.

Data Privacy and Dataset Usage. We use the publicly available *EyeTrans* dataset under its original IRB-approved consent and anonymization protocols, and further mitigate re-identification risk by releasing only aggregated statistics (Beta priors and n-gram counts), not per-participant traces.

Use of AI Assistants. The authors used AI assistants (LLMs) for minor editorial assistance, including language polishing and LaTeX formatting. All research ideation, experimental design, implementation, analysis, and claims are the authors' own.

Data Availability Statement

The distilled human-attention artifacts, including gaze priors, a schema-matching dataset sample, and reference implementations, are archived on Zenodo at `https://zenodo.org/records/17205682`; the underlying eye-tracking data originates from the EyeTrans corpus (Zhang et al., 2024) and is redistributed under its original ethical-use terms.

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.

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*.

Zihan Fang, Yifan Zhang, Yueke Zhang, Kevin Leach, and Yu Huang. 2025. Dpo-f+: Aligning code repair feedback with developers' preferences. *arXiv preprint arXiv:2511.01043*.

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.

Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. 2024. 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*.

Zachary Karas, Aakash Bansal, Yifan Zhang, Toby Li, Collin McMillan, and Yu Huang. 2024. A tale of two comprehensions? analyzing student programmer attention during code summarization. *ACM Transactions on Software Engineering and Methodology*.

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*.

Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 47–51.

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, pages 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*.

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.

Zhichao Wang, Bin Bi, Shiva Kumar Pentyala, Kiran Ramnath, Sougata Chaudhuri, Shubham Mehrotra, Xiang-Bo Mao, Sitaram Asur, and 1 others. 2024. 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*.

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*.

Xinwei Yang, Zhaofeng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025a. Elaboration: A comprehensive benchmark on human-llm competitive programming. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 59–104.

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

Yifan Zhang, Chen Huang, Zachary Karas, Dung Thuy Nguyen, Kevin Leach, and Yu Huang. 2025a. Enhancing code llm training with programmer attention. *arXiv preprint arXiv:2503.14936*.

Yifan Zhang and Kevin Leach. 2025. Leveraging human insights for enhanced llm-based code repair. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1536–1537.

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.

Yueke Zhang, Yifan Zhang, Kevin Leach, and Yu Huang. 2025b. Codegrad: Integrating multi-step verification with gradient-based llm refinement. *arXiv preprint arXiv:2508.10059*.

Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2025c. 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).

A.1 Task Performance Metrics

A.1.1 Code Translation and Completion

For Java-to-C# translation and code completion, the main tables report Exact Match, with additional code metrics available for audit:

- **Exact Match:** A prediction receives credit when the whitespace-normalized prediction and reference are identical, or when one contains the other as a complete target span:

$$\text{Exact} = \mathbb{I}(\hat{y} = y \vee \hat{y} \in y \vee y \in \hat{y}) \quad (1)$$

  where y is the ground truth, $\hat{y}$ is the generated code, and $\mathbb{I}(\cdot)$ is the indicator function. This exact-style criterion is appropriate for code tasks where generations may include the target span plus surrounding syntactic context.

- **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:

$$\text{H-Exact} = 0.5 \times \mathbb{I}(y = \hat{y}) + 0.5 \times \mathbb{I}(\hat{y} \in y) \quad (2)$$

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

$$\text{CodeBLEU} = w_1\text{BLEU} + w_2\text{BLEU}_{\text{weighted}} + w_3\text{Match}_{\text{ast}} + w_4\text{Match}_{\text{df}} \quad (3)$$

  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, the main tables report METEOR, with other standard text-generation metrics used for audit:

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

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

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

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{y_j \in y} \mathbf{x}_i^\top \mathbf{y}_j \quad (4)$$

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

B Initial Validation Results

Table 6 preserves the earlier result table. We include it as historical and diagnostic evidence that the method improved over matched baselines in the initial three-backbone study. The current main-text Table 2 should be read as the final confirmation: it uses the cleaned six-backbone evaluation, two data regimes, current quality-safe recipe selection, and the final metric choices described in Appendix A. The qualitative attention map in Figure 8 comes from the same initial diagnostic package and is retained to illustrate the attention-shift intuition.

C Implementation Details

To ensure reproducibility, we provide the configuration for EYEMULATOR and the corresponding baselines across the six-backbone evaluation.

Table 6: Initial three-backbone validation results retained for context. The main-text Table 2 reports the final six-backbone confirmation study used for the paper's main claims.

Model	Completion			Translation			Summarization			
	CodeBLEU	CrystalBLEU	H-Exact	CodeBLEU	CrystalBLEU	H-Exact	METEOR	ROUGE-1	ROUGE-L	BERTScore
StarCoder	13.90	19.41	47.66	52.07	65.07	21.11	29.06	27.47	20.78	34.04
StarCoder (EYEMULATOR)	51.98 (+38.08)	57.53 (+38.12)	79.90 (+32.24)	86.42 (+34.35)	92.61 (+27.54)	64.97 (+43.86)	33.41 (+4.35)	46.27 (+18.80)	41.09 (+20.31)	51.06 (+17.02)
Llama-3.2	19.45	26.65	50.85	71.88	82.29	53.25	27.05	24.86	18.27	33.41
Llama-3.2 (EYEMULATOR)	60.47 (+41.02)	65.90 (+39.25)	77.96 (+27.11)	83.25 (+11.37)	91.52 (+9.23)	61.02 (+7.77)	30.69 (+3.64)	44.06 (+19.20)	38.84 (+20.57)	49.49 (+16.08)
DeepSeek-Coder	13.80	18.98	49.40	58.69	69.94	24.13	22.81	21.96	15.12	28.64
DeepSeek-Coder (EYEMULATOR)	48.82 (+35.02)	54.63 (+35.65)	78.91 (+29.51)	86.34 (+27.65)	93.09 (+23.15)	65.66 (+41.53)	33.92 (+11.11)	46.86 (+24.90)	41.68 (+26.56)	51.56 (+22.92)

C.1 Model and Loss Configuration

We apply the same gaze-weighting mechanism to six backbones: `StarCoderBase-1B` (Li et al., 2023), `Llama-3.2-1B` (Meta AI, 2024; Grattafiori et al., 2024), `DeepSeek-Coder-1.3B` (Guo et al., 2024a), `Qwen2.5-Coder-1.5B`, `SmolLM3-3B`, and `Granite-Code-2B`. Each baseline uses standard causal language modeling, while EYEMULATOR scales per-token loss by gaze-derived weights w_j. The current experiments use LoRA/QLoRA adapters rather than full fine-tuning, which keeps the method lightweight and makes the six-model grid tractable.

C.2 Hyperparameters

We use task-level recipes rather than a single universal setting: completion and translation use rank-32 LoRA with weight 4, while summarization uses lower-rank variants selected from completed quality-safe sweeps. Table 7 summarizes the main shared settings.

Table 7: Main experimental hyperparameters for the current six-backbone LoRA/QLoRA evaluation.

Hyperparameter	Value
Base Models	Six 1B–3B CodeLLM backbones
Optimizer	AdamW
LR Schedule	Linear with warmup
Learning Rate	Recipe-dependent (10^{-4} to 5×10^{-5})
Effective Batch Size	16
Max Sequence Length	1024 tokens
Training Epochs	3
Weight Decay	0.01
DPO Weight γ	0.1
Base Token Weight w_{base}	1.0
Precision	bf16 / 4-bit QLoRA where supported
Hardware	NVIDIA L40S-class GPUs

C.3 Attention Prior Processing

Gaze-derived priors are injected into the loss function at the subword level. For each training example, we first run a language-specific AST parser over the source code and assign every leaf token a semantic label (e.g., `MethodDeclaration`, `IfStatement`, `VariableDeclarator`). Each label is then mapped to its posterior salience $\mathbb{E}[\theta_s]$ via the Beta priors distilled in Section 2.2. When the tokenizer splits an AST token into multiple BPE subwords, the parent's salience is propagated to each shard. The final per-token weight w_j combines this salience with a base term and an inverse-frequency correction (Section 2.4). At training time, these weights are materialized into a single tensor that is broadcast against the cross-entropy loss, so that the procedure adds negligible overhead relative to standard fine-tuning.

For the *Reading-derived* (EYEMULATOR(R)), *Writing-derived* (EYEMULATOR(W)), and *Full* variants in Table 3, the only change is the source of the Beta and n-gram statistics (reading sessions, writing sessions, or both); the model, optimizer, and training schedule remain identical across all configurations.

D Qualitative Case Studies

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

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

In this example, the baseline fails to terminate cleanly, producing a syntax error and unreachable code, while EYEMULATOR recognizes the high-level intent and uses a built-in C# method.

D.2 Case Study 2: Code Summarization

This case illustrates the "repetition loop" common in smaller LLMs: the baseline produces redundant Javadoc-style guarantees, whereas EYEMULATOR emits a single focused sentence.

E Future Work

Several directions could extend EYEMULATOR beyond the current camera-ready scope. Future work should examine whether gaze-derived priors remain effective for stronger instruction-tuned

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)

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

EYEMULATOR (Ours)

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

Figure 9: Translation case study. The baseline continues past a well-formed method and appends spurious code, while EYEMULATOR preserves the method intent and emits the idiomatic C# `IndexOf` call with the correct `short` signature.

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 10: Summarization case study. The baseline falls into a repetitive Javadoc-style pattern, whereas EYEMULATOR produces a concise summary that captures the method's behavior.

and agentic models, transfer beyond Java-centered benchmarks to broader programming settings, capture richer temporal attention signals, and improve developer-facing coding workflows.

Scaling to larger models. Our experiments focus on compact 1B–3B CodeLLMs, where full cross-task sweeps are tractable. Applying the same priors to larger instruction-tuned and agentic coding models would test whether gaze-derived signals still add value when base models already encode stronger code structure and reasoning patterns.

Expanding language and task coverage. The present study uses Java-centered gaze artifacts and CodeXGlue-style tasks. Extending the pipeline to additional languages, multi-file contexts, and repository-level tasks would clarify whether the learned signal reflects Java-specific habits or broader program-structure priors.

Modeling richer gaze dynamics. Our current priors summarize fixation and transition patterns, but eye-tracking data also contains dwell duration, regressions, scan-path uncertainty, and temporal shifts in attention. Incorporating these signals could support softer, context-dependent token weights that better distinguish quick skimming from deeper semantic inspection.

Studying interactive use. Developer studies with EYEMULATOR-adapted models would complement offline metrics by evaluating practical effects in coding workflows. Such studies could measure whether gaze-informed outputs reduce review effort, debugging time, or cognitive load during real development tasks.