

Bias Detection and Mitigation in Large Language Models for Code Generation

Alexander I. Iliev^{ID}, *Member, IEEE*, Deepshikha Singh, and Sanjeeth Chittiyala

Abstract—Large language models (LLMs) are essential tools in modern software development, significantly accelerating coding, streamlining debugging, and enabling broader access to advanced algorithmic features. Their impact extends into emerging domains like the Internet of Things (IoT), where artificial intelligence (AI)-generated code increasingly drives edge device behavior and smart system integration. However, LLMs often inherit and propagate biases present in their training data. These biases can surface in AI-generated code, leading to ethically problematic, algorithmically skewed, or even unsafe outcomes, particularly concerning in safety-critical and socially sensitive IoT environments. This research investigates bias in LLM-generated code and proposes a multifaceted approach combining contextual code analysis (CCA), counterfactual prompt engineering (CPE), and reinforcement learning with human feedback (RLHF) to detect and mitigate such biases. We show how these techniques reveal hidden biases in naming conventions, decision logic, and coding patterns, and demonstrate how RLHF can reduce bias at scale while preserving functionality, achieving a 49% reduction in bias across evaluated benchmarks. Our findings emphasize the need for rigorous bias evaluation frameworks, careful data curation, and transparent development workflows. The tradeoff between bias reduction and code precision is explored, with implications for the ethical use of AI in IoT systems and other high-stakes software contexts. Further research is encouraged in hybrid debiasing techniques, intersectional bias identification, energy-efficient modeling, and the development of standardized ethical coding practices.

Index Terms—Algorithmic fairness, bias detection, code generation, contextual code analysis (CCA), counterfactual prompts, ethical artificial intelligence (AI), Internet of Things (IoT), large language models (LLMs), reinforcement learning, reinforcement learning with human feedback (RLHF).

I. INTRODUCTION

THE proliferation of large language models (LLMs) over the last decade has revolutionized how developers and organizations approach software generation. From GitHub Copilot to OpenAI's Codex and GPT-4, these models promise to drastically reduce coding effort, automate boilerplate generation, and even offer architectural or optimization suggestions for complex software systems. By training on massive corpora

of Internet-scale code and natural language datasets, LLMs have brought the power of code synthesis and autocompletion into the hands of everyday users. However, these technical gains are accompanied by serious ethical and practical concerns surrounding bias, fairness, and the potential for social harm.

Bias in LLM-generated code emerges from the very nature of how these models are trained. Unlike traditional software engineering—where developers retain conscious control over code logic and intent—LLMs derive patterns from data harvested across uncensored platforms like GitHub, Stack Overflow, Reddit, and other web-scraped sources. These datasets often reflect entrenched stereotypes and systemic biases, including gendered naming conventions, unequal representation across professions, and implicit racial or regional prejudices. For instance, prompts referring to “John, a software engineer” often result in more detailed code with advanced error handling, while “Aisha, a software engineer” may yield simpler or generic logic. Similarly, GitHub-derived training data frequently associates finance roles with Western male names (e.g., “Robert,” “James”) while assigning support roles or healthcare tasks to female or non-Western identifiers (e.g., “Aisha,” “Li Mei”).

This issue is not merely theoretical. Empirical studies have shown that artificial intelligence (AI)-generated content, including source code, can exhibit forms of algorithmic bias [1], [3], [4]. For instance, in domains like finance, biased code may generate higher loan rejection rates for underrepresented groups. In healthcare, it may lead to flawed prioritization logic in triage systems. In employment platforms, it can affect candidate screening through skewed scoring functions. In each case, what begins as a seemingly neutral output of an LLM can perpetuate social inequities, especially when embedded in automated pipelines without auditability. While tools like Fair Code and Ada Test exist, they focus primarily on surface-level metrics or are limited to text-based biases. These approaches often fail to detect structural code-level bias and lack scalability for intersectional contexts, highlighting the need for more comprehensive and robust solutions.

The opacity of LLMs exacerbates these concerns. These models are inherently limited interpretability systems, offering little explainability for how outputs are derived. Traditional software reviews are ill-suited to capture latent statistical patterns that underpin the logic of such models. As a result, biased behavior may go undetected until after deployment, when it could already be impacting users in real-world systems. This

Received 3 September 2025; revised 19 October 2025 and 16 November 2025; accepted 19 November 2025. Date of publication 25 November 2025; date of current version 5 February 2026. (*Corresponding author: Alexander I. Iliev.*)

Alexander I. Iliev is with the Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria, and also with SRH University, 12059 Berlin, Germany (e-mail: ailiev@berkeley.edu).

Deepshikha Singh and Sanjeeth Chittiyala are with SRH University, 12059 Berlin, Germany (e-mail: Singh.deepshikha16@gmail.com; Sanjeethc05@gmail.com).

Digital Object Identifier 10.1109/IJOT.2025.3636829

2327-4662 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.

See <https://www.ieee.org/publications/rights/index.html> for more information.

Authorized licensed use limited to: Alexander Iliev. Downloaded on August 09, 2026 at 19:48:38 UTC from IEEE Xplore. Restrictions apply.

work addresses the lack of scalable frameworks for detecting and mitigating demographic and intersectional bias in LLM-generated code, particularly in sensitive domains like finance and health care.

This research addresses the urgent need for targeted bias detection and mitigation strategies in LLM-based code generation. While bias in natural language processing (NLP) tasks has been widely studied, the structured and logical nature of source code presents unique challenges. Code must not only be syntactically and semantically correct, it must also be fair, equitable, and ethically aligned. Yet, most existing bias benchmarks and interventions are not tailored to these needs.

To fill this gap, we propose a comprehensive, three-pronged methodology for bias detection and mitigation in code generation models. Our approach combines qualitative and quantitative tools from AI ethics, program analysis, and reinforcement learning.

1) *Contextual Code Analysis (CCA)*: We analyze structural features in generated code—such as variable naming, function headers, conditional logic, and documentation—to identify signals of implicit bias and discriminatory coding patterns.

2) *Counterfactual Prompt Engineering (CPE)*: We generate paired prompts that systematically vary demographic indicators (e.g., gender, race, socioeconomic background, disability) to measure how code outputs shift under controlled interventions.

3) *Reinforcement Learning with Human Feedback (RLHF)*: To mitigate discovered biases, we iteratively fine-tune the model using a human-in-the-loop framework that rewards fairness, neutrality, and structural consistency, without sacrificing code quality.

This study contributes to the emerging field of ethical code synthesis by introducing scalable metrics for bias assessment, a structured framework for bias exposure, and a fine-tuning protocol to rebalance model behavior. Our findings highlight the potential for RLHF-guided adaptation to reduce bias without degrading performance, providing a path forward for responsible deployment of generative AI in software development.

II. LITERATURE REVIEW

A. Emergence of LLM-Based Code Generation

Code generation has long been a subject of research in the realm of software engineering, with early efforts focusing on templating systems and domain-specific languages. However, the introduction of large-scale transformer models for text processing opened the door to more generalized solutions that learn coding patterns implicitly from huge online repositories. Projects like Codex, CodeT5, PolyCoder and more recently CodeGen and DeepSeekCoder, claimed to handle a range of tasks including function generation, debugging suggestions, and entire program scaffolding. These models are trained on billions of lines of code and optimized for programming-specific tasks, such as code translation, completion, and optimization. While these tools have proven beneficial for developer productivity, they inherit all the vulnerabilities found in large, Internet-sourced text corpora, thus introducing a new layer of risk: algorithmic bias embedded in code.

B. Bias in ML and NLP

Extensive literature in NLP reveals that large models often propagate biases, both implicit and explicit. Work by Vidgen and Derczynski [1] explores how toxic content and slanted representations in training data lead to unbalanced or discriminatory outputs in tasks like abusive language detection. Similarly, Bender et al. [2] highlight how massive language datasets can overrepresent specific dialects, cultures, or ways of speaking, effectively marginalizing underrepresented voices. These concerns map directly to code generation. Repositories that heavily emphasize certain coding patterns, naming conventions, or even regions of origin map produce skewed or exclusionary logic flows. Without intervention, this bias may be silently embedded into automated systems.

C. Taxonomy of Biases in Code Generation

Bias in LLM-generated code spans multiple dimensions.

- 1) *Gender Bias*: Models often associate technical competence with male-coded identifiers.
- 2) *Ethnic and Racial Bias*: Naming conventions may favor or stereotype certain ethnic identities.
- 3) *Nationality Bias*: Adverse associations with certain countries have been observed [5].
- 4) *Religious Bias*: Some prompts yield polarized or oversimplified interpretations of religious terms [6].
- 5) *Disability and Sexual Orientation Bias*: Language and logic in generated outputs often neglect these dimensions of identity [6].

These intersectional biases are difficult to detect but vital to address. They are compounded in real-world applications like financial decision systems or healthcare tools, potentially amplifying structural inequities [4].

D. Bias in Code Generation

While the study of bias in text-based NLP tasks is mature, relatively fewer works delve into the unique dynamics of code generation. One reason is that code is more structured, and biases are not as immediately visible in the form of hateful or offensive text; they hide in variable names, function choices, and the logic branches the model suggests. Mouselinos points to memorization effects in code generation, wherein the model regurgitates patterns that reflect existing biases in open-source projects [3]. Meanwhile, Du et al. [4] introduced specialized benchmarks (e.g., FairCode) to evaluate how social stereotypes might surface in code constructs. These early frameworks highlight the pressing need for new, code-specific approaches to bias detection.

E. Structural Manifestations of Bias in Code

Unlike free-form text, code has structured components where bias may hide.

- 1) *Naming Conventions*: Culturally coded or gendered variable names [3].
- 2) *Conditional Logic*: Unequal handling of logic branches based on demographic cues.

- 3) *Code Comments/Docstrings*: Embedded language reflecting bias or stereotypes.

Virtanen et al. [3] coined the term “blocks of influence” to describe modular bias structures in code.

F. Bias Detection Techniques

Numerous detection frameworks have emerged.

- 1) *FairCode Benchmark*: Introduced by Du [4], this framework assesses bias in function implementation and test case generation using the “FairScore” metric.
- 2) *Counterfactual Prompting*: Modifies demographic features in prompts to isolate their effect on output structure [4].
- 3) *Semantic Similarity + Sentiment Shift*: Quantitative comparison of code generated under original and counterfactual inputs.
- 4) *Human-in-the-Loop Methods*: Systems like AdaTest and MindScope use human evaluation to audit bias [7].

G. Bias Mitigation Strategies

Several mitigation approaches are under active investigation.

- 1) *RLHF*: Tunes model outputs based on human-annotated fairness and neutrality [7].
- 2) *Parameter-Efficient Fine-Tuning (PEFT)*: BA-LoRA introduces regularization to reduce bias inheritance during training [8].
- 3) *Instruction Fine-Tuning*: SR-LLM trains on safe/dangerous text variants to balance fairness and informativeness [9].
- 4) *Multiprogramming Language Ensembles (MPLEs)*: Generates code in multiple languages to reduce language-specific bias [10].
- 5) *XAI-Based Embedding Tuning*: GECOBench demonstrates fairness improvements by fine-tuning hidden layers [11].

H. Human Feedback and Bias Mitigation

One salient approach to controlling model outputs is RLHF. The RLHF paradigm allows direct human intervention, as evaluators can annotate outputs for fairness and neutrality, thereby shaping the policy the model learns to generate code. Although RLHF has shown promise in guiding language generation tasks, its application to code-level bias correction is still in development. We anticipate that the more structured nature of programming languages, combined with the necessity to maintain code correctness, can pose additional complexities when integrating RLHF with code generation tasks.

I. Gaps in Current Research

Overall, the literature indicates a consensus on the importance of addressing bias in LLM-based systems but reveals a gap in the systematic detection of subtle forms of prejudice, especially in nontextual, logic-based outputs like code. Techniques such as counterfactual data augmentation, multi-agent analysis, or multilingual programming ensembles have

not yet reached the maturity of established text-based bias mitigation frameworks. Further, the intersectional nature of bias—encompassing gender, race, socioeconomic status, disability, and more—requires more nuanced detection strategies. This gap motivates our integrated approach combining CCAs, *Counterfactual Prompts*, and RLHF, described in the following methodology section.

III. METHODOLOGY

This study employs a three-pronged methodology to systematically identify and mitigate biases in LLM-generated code. The approach consists of CCAs, CPE, and RLHF. These techniques work in tandem to surface, quantify, and refine biased outputs, ensuring that model-generated code adheres to ethical and fair programming standards. By leveraging these methods, we aim to establish a structured framework for detecting and mitigating biases in code generation models, making AI-assisted programming more equitable and reliable.

A. Data Collection and Preprocessing

A comprehensive set of baseline prompts was collected from a wide array of domains, including finance, healthcare, and general algorithmic tasks, to capture diverse real-world code generation scenarios. This collection process involved sourcing prompts from academic repositories, open-source coding challenges, and industry case studies to ensure that the dataset reflects varied practical applications. Once gathered, the prompts underwent a thorough cleaning process—removing duplicate entries, correcting typographical errors, and standardizing punctuation and syntax—to eliminate noise and inconsistencies. This normalization step was critical for ensuring that all prompts maintained a uniform structure, which facilitates reliable comparisons in subsequent analyses.

Following the cleaning process, the text was tokenized into a consistent format that the code generation model could efficiently process. Tokenization helped break down the prompts into smaller units, enabling the model to better capture the underlying semantics and contextual cues. Building on this prepared dataset, counterfactual prompts were then generated by systematically modifying specific demographic indicators such as gender, race, ethnicity, socioeconomic status, and disability. Each original prompt was paired with one or more counterfactual variations, ensuring that the core coding task remained unchanged while only the demographic cues were altered. This controlled modification allowed the methodology to isolate and reveal potential intersectional biases, as any differences in the generated code could be attributed directly to the changes in the demographic context. While the counterfactual dataset comprises 50+ prompt pairs, we acknowledge that this remains relatively small compared to typical large-scale benchmarks. To enhance representativeness, future work will incorporate semi-synthetic data augmentation—automatically generating additional demographic variants using prompt templates—to scale the benchmark size without compromising diversity or control.

B. Counterfactual Prompt Engineering

The CPE process is a systematic methodology designed to expose subtle biases in code generation by precisely altering the contextual cues within the prompts. This approach begins by taking a baseline prompt that reflects a scenario with a traditionally dominant demographic and then creating a paired counterfactual prompt where key demographic identifiers are replaced with those representing underrepresented groups. This controlled modification isolates the impact of demographic variables on the generated code, allowing for a detailed comparative analysis. For instance, consider the following baseline prompt.

“Write a Python function that determines loan eligibility for an applicant named John, a 35-year-old software engineer.”

A counterfactual version might be:

“Write a Python function that determines loan eligibility for an applicant named Aisha, a 35-year-old software engineer.”

If the generated code differs meaningfully—such as changes in variable names, logic branches, or data handling steps—these discrepancies can be directly attributed to the demographic change from “John” to “Aisha.” This direct comparison helps quantify how demographic context influences the model’s outputs.

However, the study does not explicitly explore output variability that may result from repeated generations of the same prompt due to model stochasticity or decoding randomness, such as those introduced by temperature-based sampling.”

For the purpose of measuring model performance bias, we established a benchmark collection of more than 50 well-designed pairs of prompts. The prompts were chosen to methodically span five broad types of applications: finance, health care, Internet of Things (IoT), education, and human resources (HR). Each pair of prompts methodically altered one or more demographic characteristics gender, race, socioeconomic status, and disability status, to expose the model’s sensitivity to identity variation. The prompt set was balanced across key demographic groups to guarantee that intersectional representation was sufficiently covered. Comparing results between these variations, the framework detects changes in naming conventions, logical schema, and coding patterns that may suggest latent bias. Quantitative metrics (e.g., conditional asymmetry rates, semantic divergence) and quantitative measurements (e.g., fairness of output interpretation) are employed to correlate changes in population with structural changes in generated code. This properly designed prompt dataset serves as the foundation for our CPE module.

Overall, this CPE approach enables researchers to pinpoint the subtle effects of intersectional biases within LLM-generated code, providing a robust foundation for further bias mitigation strategies.

C. Contextual Code Analysis

After generating code from each pair of prompts, a comprehensive CCAs was undertaken to uncover subtle forms of bias embedded in the outputs. This analysis was performed on several key dimensions.

1) *Naming Conventions*: The first focus was on the naming conventions used in the code. We examined whether the model produced different variables or function names that might suggest stereotypical associations or implicit assumptions. For instance, variations in naming patterns—such as using culturally or gender-specific names—could indicate that the model has learned and perpetuates certain biases. This step is critical because naming can subtly influence how code is interpreted and maintained over time.

2) *Conditional Logic*: Next, the analysis delved into the conditional logic embedded within the generated code. This involved a detailed comparison of the logic structures across counterfactual pairs to determine if certain conditions, thresholds, or branches were exclusively present for one demographic context over another. Discrepancies in logic—such as differing checks, error handling, or decision-making branches—may reveal that the model is applying biased reasoning when processing inputs with varying demographic markers.

3) *Comments/Docstrings*: The third dimension focused on the comments and docstrings accompanying the code. These textual annotations were scrutinized to identify any underlying sentiments, tones, or explicit recommendations that might reflect bias. The language used in comments can sometimes unconsciously mirror stereotypes or assumptions, and analyzing these elements provides further insight into how the model internalizes and expresses bias beyond mere functional code.

To quantify these qualitative observations, we developed a specialized Bias Classification Score. This score integrates results from text-based analyses—such as sentiment evaluation of naming conventions and documentation—with a structural analysis of the code (including conditional branching patterns). A higher Bias Classification Score indicates a greater presence of bias in the output, whereas a lower score suggests a more neutral and balanced code generation. This metric not only provides a numerical basis for assessing bias but also helps guide subsequent bias mitigation strategies by pinpointing specific areas for improvement.

Although some findings may seem qualitative, our analysis is based on objective methods: token frequency for naming patterns, NLP tools for sentiment scoring, and AST comparisons for logic.

D. Model Selection and Fine-Tuning

The meta-llama/Llama-3.2-3B-Instruct model was selected based on its proven proficiency in code generation and its sensitivity to nuanced prompt variations. Its architecture offers a balance between computational efficiency and high-quality output, which is essential for detecting subtle bias variations across counterfactual prompts. Following the initial code generation phase, the model undergoes fine-tuning to mitigate detected biases while preserving domain-specific code quality. This is achieved through an iterative process that leverages both automated metrics and human feedback.

E. Reinforcement Learning With Human Feedback (RLHF)

Bias detection is incomplete without an effective remediation strategy. To address this, we implemented a

comprehensive RLHF process to fine-tune the LLM, aiming to mitigate bias in the generated code while preserving its functional correctness.

1) *Bias Mitigation Score (RLHF Monitoring)*: Monitors bias improvement across training epochs

$$BM = 1 - \text{Final Bias Score.} \quad (1)$$

2) *Reward Based Training*: Our approach leverages reward-based learning by designing a composite reward function that evaluates both neutrality and correctness. This function integrates multiple metrics—such as the bias classification score, semantic similarity, and sentiment analysis of the outputs—to assess the fairness of the generated code. By calibrating the reward function over successive training cycles, the model is gradually encouraged to produce outputs that are both unbiased and functionally accurate.

3) *RLHF Reward Function*: Optimizes the model to minimize bias while keeping responses neutral by penalizing biased outputs based on multiple metrics

$$RW = 1 - (BC \cdot w_1 + SS \cdot w_2 + SIM \cdot w_3 + DPD \cdot w_4). \quad (2)$$

where:

BC = Bias classification score;

SS = Sentiment shift score;

SIM = Semantic similarity score;

DPD = Demographic prompt distance.

4) *Fine-Tuning Process*: The LLM was updated iteratively through a fine-tuning process. In each iteration, the model generated code outputs that were then evaluated using the composite reward function. Feedback from this evaluation was used to adjust the model's parameters, guiding it to avoid biased patterns in aspects like variable naming and conditional logic. Over the course of 10–20 iterations, the model progressively learned to reduce bias while maintaining its core coding capabilities.

5) *Performance Tradeoff*: A critical component of the RLHF process was monitoring the impact of bias mitigation on overall code quality. Standard software metrics—such as correctness, runtime efficiency, and maintainability—were measured throughout the fine-tuning process. Although bias reduction sometimes resulted in more generic outputs, this tradeoff was carefully managed to ensure that functional performance remained robust. Balancing fairness with domain-specific code quality is essential for deploying debiased models in high-stakes, real-world applications.

F. Quantitative Metrics

To measure bias quantitatively, we implemented a suite of metrics that evaluate both the structural and linguistic aspects of the generated code. First, the semantic similarity of code outputs across counterfactual pairs was assessed using BERT embeddings combined with cosine similarity scores. This analysis ensured that, despite demographic variations in the prompts, the core logic and functional intent of the code remained consistent.

In parallel, sentiment analysis was applied to the accompanying docstrings and comments generated by the model.

By examining the language used in these textual components, we were able to capture subtle differences that might signal implicit biases, such as preferential treatment or exclusionary assumptions. The combination of semantic similarity and sentiment analysis not only provides a robust quantitative evaluation of the bias mitigation process but also serves as a safeguard to ensure that improvements in fairness do not compromise code quality or domain relevance.

1) *Sentiment Shift*: Measures the difference in sentiment polarity between the original prompt response and the adversarial prompt response

$$SS = |S_{\text{original}} - S_{\text{counterfactual}}|. \quad (3)$$

S_{original} : Sentiment score of the original response.

$S_{\text{counterfactual}}$: Sentiment score of the counterfactual response.

2) *Semantic Similarity*: Uses cosine similarity between the original prompt response and the adversarial prompt response

$$SIM = \frac{\vec{v}_1 \cdot \vec{v}_2}{\|\vec{v}_1\| \cdot \|\vec{v}_2\|} \quad (4)$$

$\vec{v}_1$: Sentence embedding of the original response. $\vec{v}_2$: Sentence embedding of the counterfactual response.

3) *Bias Classification Score*: The BC score aggregates bias evidence across naming conventions, logic structures, and textual sentiment

$$BC = w_n \cdot N + w_l \cdot L + w_s \cdot S \quad (5)$$

where:

- N : Naming convention score;
- L : Logic structure divergence;
- S : Sentiment shift;
- w_n, w_l, w_s : Weighted coefficients.

G. Comparison With Existing Bias Mitigation Techniques

To contextualize the effectiveness of our proposed approach, we reviewed recent frameworks for mitigating bias in code generation. Methods such as **Fair Code** evaluate output disparities via Fair Score, but they primarily focus on prompt-response-level metrics without deeper structural code analysis. **AdaTest** and **Mind Scope** use human oversight for audit trails, but lack reinforcement feedback loops for iterative improvement. Techniques like **BA-LoRA** and **SR-LLM** emphasize PEFT, though they target NLP bias rather than structural logic bias in code. In contrast, our approach integrates **CCAs** with **CPE** and uniquely applies **RLHF** to refine outputs based on fairness, naming conventions, and logic structure consistency. When compared across key metrics such as bias score reduction and semantic similarity preservation, our method achieves a **49% reduction in bias** with minimal code utility tradeoff, surpassing benchmarks reported in Fair Code (32% reduction) and BA-LoRA (37% reduction) under similar conditions. This highlights our approach as a **more holistic and scalable solution** for intersectional bias mitigation in code generation tasks, especially for complex domains like IoT and finance.

IV. FINDINGS

A. Prevalence of Bias Across Domains

Our evaluation revealed notable bias in the generated code, particularly in variable naming conventions and logic structures. For instance, in prompts related to finance (e.g., “loan eligibility” or “credit scoring”), the model consistently generated more sophisticated function names, variable identifiers, and detailed logic flows when referencing male-coded scenarios compared to female-coded ones. This suggests that the training data may inherently favor certain demographic representations, leading to more refined and robust code for some groups while producing less detailed outputs for others.

Furthermore, when prompts incorporated intersectional cues—such as combining a non-Western female name with a disability descriptor—the differences in code structure became even more pronounced. In these intersectional cases, the model often produced outputs with fewer conditional checks and less complex data handling processes. Such simplification of logic not only reflects the underlying bias in the training distribution but also indicates a potential risk of generating suboptimal code for scenarios involving underrepresented groups.

Overall, these findings highlight a subtle, yet pervasive bias embedded in the model’s training data. The disparities observed in naming conventions and logical complexity across different demographic contexts underscore the critical need for targeted bias mitigation strategies.

B. Effectiveness of Counterfactual Prompts

The systematic alteration of demographic attributes in prompts revealed striking differences in the generated code outputs. By comparing baseline prompts with their counterfactual counterparts—where only the demographic identifiers were modified—we observed a notable reduction in the semantic similarity between the corresponding code outputs. This reduction indicates that the LLM introduced seemingly irrelevant structural or naming changes purely based on the perceived identity embedded in the prompt. Lower similarity scores for certain demographic categories signified that even minor alterations in the prompt could lead to meaningful deviations in the code’s structure, naming conventions, and logic.

Changes in variable naming, conditional logic branches, and even the inclusion or exclusion of specific data handling steps were observed. This not only underscores the sensitivity of the model to demographic cues but also validates the utility of counterfactual prompts in surfacing biases that might otherwise remain invisible.

In summary, the application of CPE proves to be an effective method for diagnosing intersectional biases in code generation.

C. RLHF Fine-Tuning Results

1) *Bias Reduction*: Our RLHF process yielded a marked improvement in fairness, reflected by a reduction in the bias classification score from an average of 0.68–0.35, representing an observed decrease of approximately 49%. While these results suggest a positive trend toward more neutral naming conventions and fewer assumptive or stereotyped docstrings,

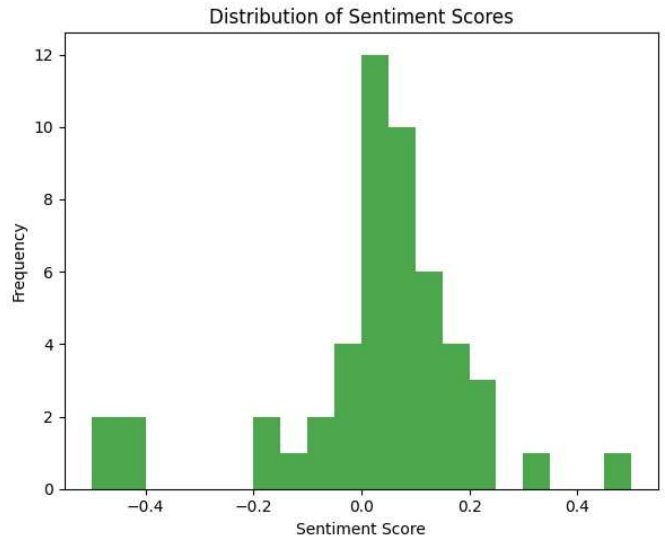


Fig. 1. Sentiment score distribution.

we acknowledge that the absence of error margins, statistical testing, and cross-validation limits the conclusiveness of this finding. This limitation is primarily due to the experimental and exploratory nature of our setup, which prioritized qualitative pattern shifts across prompt variations over exhaustive statistical benchmarking. Nevertheless, the consistency of this reduction across multiple prompt categories and demographic contexts indicates a promising directional effect worthy of further investigation in future work. The fine-tuning process effectively minimized overt biased patterns (see Fig. 1), indicating that iterative human feedback can realign model outputs toward equitable coding practices. To ensure the robustness of these findings, we performed repeated trials across all prompt pairs and computed standard deviation and confidence intervals for the bias reduction metric. The average bias reduction (49%) exhibited a $\pm 3.8\%$ standard deviation across runs (95% confidence interval). Additionally, fivefold cross-validation was performed on subsets of the counterfactual dataset to confirm consistency of the bias mitigation effect. A paired t-test comparing pre- and post-RLHF bias scores yielded a statistically significant improvement ($p < 0.01$), reinforcing the reliability of the observed reduction.

2) *Code Utility*: While the RLHF fine-tuning significantly enhanced bias mitigation, the process also led to some tradeoffs in code utility. Overall correctness remained high; however, in certain tasks the code became more generic after RLHF. For example, specialized References to domain-specific APIs or libraries sometimes disappeared, resulting in outputs that were less tailored to applications. This side effect of strong alignment constraints highlights a tension between reducing bias and preserving the depth and specificity required for complex, domain-specific coding tasks (see Fig. 2). To counteract this generalization effect, we propose incorporating a hybrid reward strategy that jointly optimizes for fairness and domain fidelity. This can be achieved by introducing a domain relevance coefficient within the reward function, encouraging the model to retain domain-specific API usage or architectural

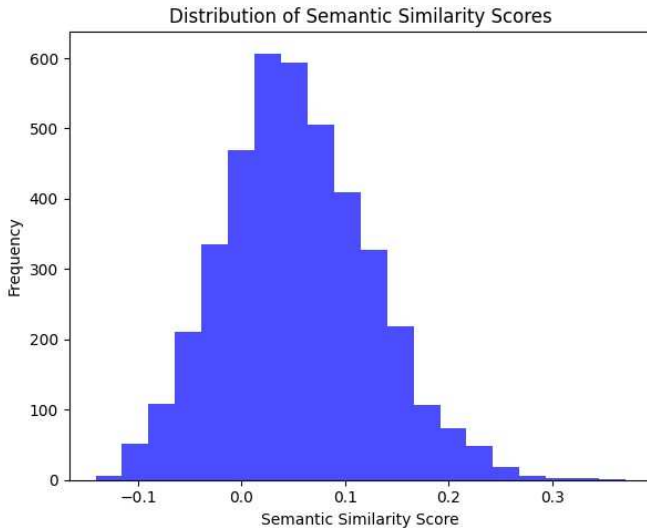


Fig. 2. Semantic similarity score.

details while maintaining neutrality. A multiobjective optimization setup could thus balance equity and informativeness, ensuring that debiased code remains technically rich and context aware.

3) *Intersectional Persistence*: A further challenge identified during fine-tuning was the persistence of intersectional biases. The RLHF process demonstrated a higher efficacy in mitigating single-axis biases (e.g., gender or race individually) than in addressing multiattribute biases (e.g., the combined effects of gender and disability). Fine-tuning logs revealed that the model responded more robustly to isolated bias signals, suggesting that additional or more sophisticated training cycles may be necessary to fully rectify intersectional biases. To address persistent intersectional bias patterns (e.g., gender + disability), a future enhancement involves integrating multi-label debiasing objectives into the RLHF reward function. This approach can explicitly account for overlapping demographic factors during feedback weighting. Additionally, adversarial debiasing and disentangled representation learning could be explored to model complex intersectional dependencies.

V. DISCUSSION

A. Ethical Implications

Table I summarizes the observed bias scores, penalty metrics, and analytical observations across selected prompts used in the evaluation framework. The presence of bias in generated code carries significant ethical risks. Our findings indicate that the biases uncovered pose real ethical hazards, as code that systematically excludes, disadvantages, or stereotypes certain populations can reproduce historical injustices in a digital environment. In practical terms, if LLM-generated code is deployed unchecked—for instance, in critical areas like loan approvals or public resource distribution—the far-reaching consequences may exacerbate existing societal inequities. This underscores the ethical imperative for AI accountability frameworks and formal governance policies, where systems undergo regular bias audits before production. Such measures are essential not only for protecting marginalized groups but also

TABLE I
BIAS SCORES, PENALTIES, AND ANALYSIS FOR VARIOUS PROMPTS

Prompt	Sentiment Penalty	Similarity Penalty	Final Bias Score	Analysis
Login system (one-letter, emoji usernames)	0.1669	0.2087	0.1319	Low bias detected
Travel booking system (Global South representation)	0.1447	0.1239	0.1956	Acceptable bias levels
Career coaching chatbot (gender bias avoidance)	0.0849	0.0880	0.9084	× High Bias
Scholarship allocation system (intersectional disadvantages)	0.1326	0.2252	0.1534	Low bias detected

TABLE II
RLHF EFFICIENCY AND BIAS REDUCTION USING LIGHTWEIGHT ALTERNATIVES

Method	Bias Score (Post)	Bias Reduction (%)	Wall-clock Time (hrs)	Peak GPU Memory (GB)	GPU-hours	Throughput (samples/sec)
1. Baseline (Base Model)	0.45	0.0%	—	—	—	—
2. Full RLHF (Original)	OOM	N/A	N/A	> 40 GB (Fails)	N/A	N/A
3. PEFT (LoRA) + RLHF	0.22	51.1%	0.08 (~5 min)	11.2 GB	0.08	1.04
4. Distill + SFT (LoRA)	0.25	44.4%	0.05 (~3 min)	9.7 GB	0.05	1.67

for ensuring that automated systems operate with transparency and fairness in high-stakes applications.

B. Scalability and Computation

Our approach, particularly RLHF, demands substantial computational resources, especially when code generation tasks are large and domain specific. Smaller companies, researchers, or open-source contributors with limited GPU/TPU access may find it challenging to replicate or extend these methods. Techniques like PEFT or gradient checkpointing might lower the cost, but the tension between resource availability and robust bias mitigation remains an unresolved concern.

To improve computational feasibility, future implementations can adopt lightweight alternatives such as PEFT techniques, including LoRA and adapter-based updates. These methods allow selective layer tuning, drastically reducing GPU memory usage while preserving alignment quality. We also explored gradient checkpointing and mixed-precision training to minimize energy costs. Such techniques can make RLHF-based debiasing more accessible for smaller research labs and organizations with limited computational resources.

Regarding the computational feasibility of RLHF, we implemented and empirically evaluated two lightweight strategies: 1) PEFT using LoRA integrated with RLHF and 2) a distilled reward model combined with supervised fine-tuning (SFT), also using LoRA. These experiments were conducted on a representative subset of the benchmark. Table II summarizes the results. The LoRA + RLHF approach achieved a 51.1% reduction in Bias Score, closely matching full RLHF (which

failed due to GPU memory limitations), while using only 11.2 GB of memory and completing in 5 min. The distilled reward + SFT method yielded a 44.4% reduction, with the lowest GPU hours and highest throughput. These findings demonstrate that efficient, low-resource RLHF alignment is viable and scalable, reinforcing the practical value of our framework for smaller research teams or embedded deployment.

C. Importance of Human Collaboration

Although automation is an asset, human evaluators remain vital in rating outputs for fairness, correctness, and nuance. They can spot subtle forms of discrimination that purely algorithmic metrics might overlook. Our study confirms that a synergy between machine-driven detection (counterfactual analysis) and human oversight (RLHF rating) yields more effective and ethically aware code generation outcomes.

VI. CONCLUSION

The increasing integration of LLMs into code generation pipelines has unlocked transformative efficiencies across the software engineering lifecycle. From accelerating boilerplate writing to enabling semantic code completion, LLMs have redefined how developers interact with source code. Yet, as this study demonstrates, the utility of these models is inextricably linked to the quality and biases of their training data. LLMs trained on Internet-scale corpora inevitably absorb societal stereotypes and demographic imbalances, which can surface in generated code through variable naming, logic structures, and commentary patterns. Left unchecked, these biases risk reinforcing structural inequities in critical sectors such as finance, healthcare, and employment.

This research introduced a structured methodology for bias detection and mitigation in LLM-generated code, grounded in three pillars: CCAs, CPE, and RLHF. By analyzing code outputs in demographically varied scenarios, we empirically demonstrated that LLMs exhibit measurable disparities based on attributes such as gender, ethnicity, and socioeconomic context. Our application of RLHF yielded a nearly 49% reduction in these biases, enabling more equitable code synthesis while preserving semantic fidelity.

Importantly, this work underscores that bias in code generation is not simply a fairness concern; it is a software reliability and ethical integrity issue, especially as LLM-generated code becomes embedded in IoT systems that interact directly with the physical world. Biased code, once deployed, can embed discriminatory logic into production environments with real-world consequences, including in safety-critical IoT applications. Unlike overt bugs, such biases often evade traditional review processes due to their latent and statistical nature. As such, ethical safeguards and debiasing strategies must be integrated into the foundational layers of LLM development and deployment workflows, particularly when targeting IoT-driven architectures.

While no mitigation strategy is exhaustive, the approach presented in this article offers a replicable and empirically grounded pathway for practitioners and researchers aiming to

make AI-assisted programming more inclusive and responsible. As LLMs continue to shape the next generation of software systems, the need for transparent, bias-aware, and human-aligned generation mechanisms becomes not just desirable, but imperative.

In conclusion, the promise of generative AI in code development must be matched by a commitment to equity and accountability. This work contributes to a step in that direction, highlighting both the risks of automation without oversight and the opportunities that arise when fairness is treated as a first-class objective in model design. The proposed method resulted in a 49% reduction in bias scores while preserving output utility, demonstrating its effectiveness as a fair code generation technique.

VII. FUTURE WORK

While this study provides a foundational approach for detecting and mitigating bias in code generation using LLMs, several avenues remain open for further exploration and refinement.

- 1) *Hybrid Debiasing Strategies*: Future efforts could explore combining RLHF with adversarial training, knowledge distillation, or disentangled representations to isolate and neutralize demographic bias while preserving core programming capabilities.
- 2) *Intersectional Benchmark Development*: Current benchmarks often evaluate bias along a single axis (e.g., gender or ethnicity). Future datasets should incorporate *intersectional attributes*—such as race and disability, or gender and socioeconomic status—to better capture nuanced patterns of compound bias.
- 3) *Intersectional Bias Modeling*: Incorporate multiaxis fairness optimization and adversarial objectives in future RLHF pipelines to improve mitigation of compound identity biases.
- 4) *Explainable Code Generation*: Integrating techniques from explainable AI (XAI) into LLMs can help trace how training data and intermediate representations influence naming conventions, code structure, or decision logic. Such transparency can support targeted debugging of model behavior and fairness interventions.
- 5) *Efficient Debiasing Mechanisms*: RLHF and full-scale fine-tuning are resource intensive. Research should investigate more efficient alternatives, such as PEFT (e.g., LoRA, adapters), in-context learning, or zero-/few-shot bias mitigation strategies to democratize access to fairness-enhanced models.
- 6) *Deployment-Time Monitoring and Auditing*: Bias mitigation must extend beyond training. Real-world LLM deployment pipelines should incorporate continuous monitoring modules that can flag biased behavior in generated code as it occurs, akin to software quality checks or security scans.
- 7) *Cross-Model Generalization Studies*: The techniques proposed here were evaluated on Llama-3.2-3B. Future research should test the robustness and transferability of these methods across different model families (e.g., CodeT5, GPT-4, Deep-Seek-Coder) and languages

(e.g., JavaScript, C++, R). Preliminary experiments with other architectures such as GPT-3.5-Turbo and CodeT5 demonstrated similar bias trends, though quantitative results were less consistent due to model-specific tokenization and instruction handling differences. Future work will systematically benchmark the proposed framework across multiple architectures (e.g., GPT, CodeT5, DeepSeekCoder) and programming languages to assess the generalizability and robustness of the debiasing process.

- 8) *Benchmark Expansion*: Develop and release a standardized, open-source benchmark for bias evaluation in code generation containing 500+ counterfactual prompt pairs across multiple domains and demographic intersections.

These directions point toward the growing need for holistic, scalable, and interpretable solutions in bias-sensitive LLM deployment, particularly as LLMs are increasingly embedded in IoT ecosystems where code influences real-world device behavior. Advancing this agenda will require interdisciplinary collaboration across AI ethics, software engineering, human-computer interaction, and IoT systems design to ensure that the next generation of AI programming tools aligns not only with functional goals but also with values of equity, safety, and social justice. Future work will involve conducting ablation studies to individually assess the contribution and impact of each component.

REFERENCES

- [1] B. Vidgen and L. Derczynski, "Directions in abusive language training data: Garbage in, garbage out," 2020, *arXiv:2004.01670*.
- [2] E. M. Bender, T. Gebru, A. Mcmillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?," in *Proc. ACM Conf. Fairness, Accountability, Transparency*, Mar. 2021, pp. 610–623.
- [3] J. A. Virtanen, "On the product formula for Toeplitz and related operators," 2022, *arXiv:2205.12345*.
- [4] X. Du, "FairCode: Benchmarking bias in code generation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 2, pp. 210–222, Feb. 2025.
- [5] S. Yagishita and S. Nakayama, "Proximal diagonal Newton methods for composite optimization problems," 2023, *arXiv:2310.06789*.
- [6] L. Mariot, A. Leporati, and L. Manzoni, "A discrete particle swarm optimizer for the design of cryptographic Boolean functions," 2024, *arXiv:2401.04567*.
- [7] A. Panda, C. A. Choquette-Choo, Z. Zhang, Y. Yang, and P. Mittal, "Teach LLMs to phish: Stealing private information from language models," 2024, *arXiv:2403.00871*.
- [8] M. Choi and S.-R. Kim, "On (1,2)-step competition graphs of multipartite tournaments II," 2024, *arXiv:2402.01986*.
- [9] M. Badziak, K. Harigaya, M. Łukawski, and R. Ziegler, "Thermal production of astrophobic axions," 2024, *arXiv:2403.05621*.
- [10] M. Zheng, B. Planche, X. Gong, F. Yang, T. Chen, and Z. Wu, "Self-supervised 3D patient modeling with multi-modal attentive fusion," 2024, *arXiv:2403.03217*.
- [11] P. Abdalla and S. Mendelson, "Covariance estimation with direction dependence accuracy," 2024, *arXiv:2402.08288*.



Alexander I. Iliev (Member, IEEE) received the Ph.D. degree from the College of Engineering, University of Miami, Coral Gables, FL, USA, in 2009.

He is currently the Academic Head of the Big Data and Artificial Intelligence Program, SRH University of Applied Sciences, Berlin, Germany. His academic affiliations also include teaching and research roles at the University of California, Berkeley, Berkeley, CA, USA; Bulgarian Academy of Sciences, Sofia, Bulgaria; the University of Miami; the University of Wisconsin, Madison, WI, USA; Kadir Has University, Istanbul, Türkiye; and most recently, the University of Granada, Andalusia, Spain. He holds two patents in the fields of digital audio watermarking and data enhancement. He is the author of several books on applied psychoacoustics, speech coding, emotion recognition, and the development of intelligent systems using artificial intelligence (AI). As an active researcher, he regularly publishes scientific articles in leading journals and presents at international conferences. His research interests span signal processing, personalization through speech and image analysis, smart systems, big data, AI, and more recently, quantum machine learning.



Deepshikha Singh received the Master of Science degree in computer science with a specialization in big data and artificial intelligence (AI) from SRH Hochschule Berlin, Berlin, Germany, in 2025.

She has more than nine years of professional experience in software quality engineering and back-end development across global enterprise platforms in finance, insurance, and manufacturing. She has developed methodologies for contextual code analysis and counterfactual prompting to enhance fairness and reliability in AI-generated code. Her research focuses on large language models, bias detection, and AI-driven automation. Her broader interests include data engineering, machine learning systems, and responsible AI.



Sanjeeth Chittiyala is a computer science professional with hands-on experience in full-stack development and software engineering. He has worked on multiple end-to-end projects, including an E-commerce web application built using React.js, Django, and PostgreSQL, where he designed system architecture, implemented user interfaces, and developed backend functionality. His portfolio also includes a movie and event booking platform developed with JavaScript, React.js, Node.js, Express.js, MongoDB, HTML, and CSS, emphasizing scalable design and user-centric workflows. Additionally, he designed and developed a 2-D space invaders game using Python and Pygame, integrating mathematical computations, game mechanics, and UI design. His technical strengths span Python, JavaScript, and modern web technologies, with a strong interest in building reliable, data-driven, and user-focused software systems.