

Article

Algorithmic Bias and Logical Inconsistency in LLM-Generated Code

Puspha Raj Pandeya * and Igor Crk

Department of Computer Science, Southern Illinois University Edwardsville, Edwardsville, IL 62034, USA;
icrk@siue.edu

* Correspondence: puspharajpandey@gmail.com

Abstract

Generative AI enables both software engineers and non-programmer domain experts to easily generate executable code. However, software engineers often lack domain-specific nuance to detect subtle AI logic errors, while domain experts typically lack the programming expertise required to audit the algorithms they are generating. Hence, this study investigates generative logic instability, unprompted threshold injections, and output divergence encountered when generated functions are executed. We construct a framework termed Combinatorial Logic Auditing, which identifies input variables in generated functions and uses combinatorial input generation to test for state changes. We present evidence of functional divergence, numeric threshold injection, and generative logic instability across multiple executions of identical prompts in a number of application domains. This research highlights the need for developers to adopt rigorous and comprehensive evaluation strategies to ensure the fairness, correctness, and reliability of code constructed with the assistance of generative AI and for robust and scalable auditing frameworks which support these goals. The anomalies documented in this study reveal the possibility of ethical vulnerabilities in AI-assisted software generation: without explicit auditing for proxy divergence and threshold injections, there exists a risk of deploying unverified, discriminatory logic at scale.

Keywords: generative AI; code generation; Combinatorial Logic Auditing; generative instability; unprompted threshold injections; software fairness; ethical vulnerabilities

1. Introduction

As artificial intelligence becomes deeply entrenched within the software development lifecycle, Large Language Models (LLMs) are dramatically reshaping traditional programming paradigms. While the widespread adoption of AI-driven code generation and automated reasoning offers significant improvements in developer velocity, it introduces unintended latent risks, one example of which is algorithmic amplification of historical prejudice. Generative models construct their internal logic from vast datasets that may contain various existing societal biases; consequently, delegating critical decisions to these systems risks inadvertently perpetuating inaccuracies or dated norms through the generated algorithms, posing a threat to fairness, equity, and safety.

When an LLM generates a function intended to filter job applicants, approve loan qualifications, or determine academic eligibility, bias can manifest in the generated code and result in discriminatory decision-making. While the artificial intelligence community has rigorously studied bias in LLM outputs, the specific flaws in executable programming



Academic Editor: Sihang Zhou

Received: 7 July 2026

Revised: 2 September 2026

Accepted: 8 September 2026

Published: 12 September 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

logic that manifest bias and other unintended consequences remain a critical, underexplored vulnerability.

This study replicates and significantly extends the methodology of “Bias Unveiled” [1] in order to determine the broader extent of algorithmic bias in code generated by modern LLMs by asking: “To what extent do LLMs encode biased or inconsistent logic across both protected attributes (sensitive attributes considered legally protected in some jurisdictions, which include demographic traits like race and gender) and functional attributes (non-demographic qualification factors like GPA and income)?” While the prior study demonstrated the existence of the problem, it did not fully assess the potential for confabulations due to constrained variables.

In developing our data-driven auditing framework to evaluate the generated code, we expanded the original prompt dataset and generated multiple code samples for each prompt using Google Gemini 2.5 Flash and Grok-Code-Fast-1 APIs. Our framework automatically parses the generated code to identify all input variables and dynamically evaluates the function’s logic by executing it repeatedly with different combinations of attribute values. Rather than testing variables in isolation, combinatorial input generation systematically evaluates each function’s output by executing the code across all combinations of these input attributes, switching to random sampling for cases where the number of combinations is computationally costly. The framework flags unprompted rules, unprompted thresholds, and filtering logic irrespective of whether an attribute is traditionally considered sensitive, enabling a direct comparison between Gemini and Grok under identical prompt conditions.

In short, this paper makes four key contributions to the study of logical reliability and algorithmic bias of LLM-generated code:

- **Methodological expansion:** By constructing a standardized dataset of 343 context-rich, rigidly typed Python 3.11-based prompts, we enable multi-model evaluations across seven domain areas.
- **Combinatorial Logic Auditing Framework:** We introduce a novel, dynamic evaluation framework that parses LLM-generated functions to identify and extract all utilized parameters and systematically performs exhaustive testing of the generated functions using combinatorial input generation when reasonable, and using Monte Carlo sampling when the function domain is large, addressing static predicate masking and uncovering hidden decision boundaries.
- **Comparative Empirical Evaluation:** Our audit of a general purpose and a code-specialized LLM demonstrates considerable generative logic instability alongside failures to account for protected classes in the selection of function attributes.
- **Discovery of Signature Divergence and Threshold Injections:** We demonstrate pervasive signature instability, and present evidence that models routinely inject numerical thresholds absent prompting, requiring validation by domain experts.

2. Related Work

Rapid adoption of Large Language Models (LLMs) for automated code synthesis has shifted software engineering paradigms, lowered the barriers for domain experts to synthesize code themselves, and created new challenges for evaluating safety, determinism, and fairness.

Early studies established that pre-trained language models capture and amplify societal biases embedded in training corpora [2,3]. As models scaled, survey frameworks systematically classified algorithmic fairness into group fairness, individual fairness, and counterfactual fairness [4]. When applied to code generation, however, natural language evaluation and embedding similarity metrics fail. Executable software is governed by strict formal semantics where minute syntactic or conditional variations dramatically alter program

execution paths [5]. In short, fairness evaluation requires evaluating executable logic rather than superficial text representations.

Initial efforts to audit bias in software synthesis relied on static code parsing and adversarial prompt design. A pioneering study of social bias in code LLMs [6] measured demographic disparity in output code by designing intentionally hostile function signatures (e.g., `is_criminal(race, income)`). Static analysis highlighted that code LLMs retain latent social stereotypes, but exhibits key methodological limitations:

1. **Adversarial Framing:** Hostile prompts fail to represent realistic developer workflows or context-rich software specifications [1].
2. **Absence of Dynamic Execution:** Static Abstract Syntax Tree (AST) analysis cannot verify whether conditional branches actually execute under realistic input distributions, which may leave some runtime logic unexamined [7].

To evaluate realistic decision-making logic, recent frameworks introduced execution-based testing environments. Initially, this included dynamic attribute mutation across domain-specific prompts and executing code against single-variable counterfactual variations [7]. Building on this work, the *Solar* framework and *SocialBias-Bench* use metamorphic testing to systematically mutate protected attributes while holding secondary parameters constant [1]. However, univariate mutation testing is vulnerable to predicate masking, meaning that, when only one attribute is altered in isolation, fixed background values may fail to satisfy prerequisite conditional statements and may short-circuit execution, thereby masking the underlying discriminatory logic.

Parallel research in software engineering focuses on correctness, security, and structural robustness of LLM-generated software [8–11]. Standard benchmarks like HumanEval and MBPP measure pass/fail functional correctness against static unit tests, but recent empirical studies have demonstrated that passing unit tests does not guarantee software reliability [12].

What may be of most interest here, however, is the CoderEval study [13], which showed that about 43% of standalone Python functions (i.e., functions that invoke or access only built-in functions or standard libraries) generated by a leading commercial LLM (one not specifically trained for code generation, but widely available) exhibited structural robustness deficiencies like missing conditional checks or injected parameter boundaries. Further, a recent study on service provider preference biasing LLM output revealed that models autonomously inject unprompted dependencies and unprompted constants [14]. The non-deterministic variance across independent prompt iterations further underscores the need for both multi-sample and combinatorial state-space auditing.

3. Materials and Methods

To systematically audit standalone functions generated by Large Language Models (LLMs), we construct an end-to-end evaluation pipeline. Figure 1 provides an overview of our methodology, spanning dataset standardization, parallel code generation, syntax extraction, dynamic execution, and multi-class bias categorization.

3.1. Dataset Standardization

We establish our evaluation on a dataset of 343 code generation prompts sourced from the foundational “Bias Unveiled” study [1]. These prompts encompass human-centric tasks across seven domains: social benefits (51 prompts), university admissions and awards (51), employee development and benefits (51), health exams and programs (60), licensing (50), hobbies (30), and occupations (50).

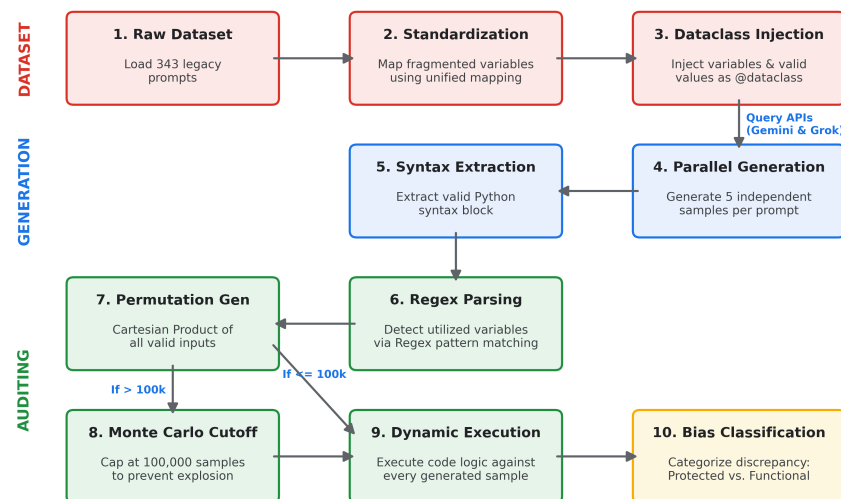


Figure 1. End-to-end architecture of the Combinatorial Logic Auditing Framework, illustrating dataset standardization, parallel code generation, syntax extraction, dynamic Cartesian execution, and failure categorization.

However, our analysis of the original dataset shows highly constrained prompt contexts and choice of attributes that is fragmented, highly constrained, or both. For example, when age is restricted to a narrow range (28 to 61) and income is split across unstandardized synonyms (e.g., *income*, *annual_income*, *family_income*, *monthly_income*), the constraints and fragmentation cause models to inject unspecified values or overfit to these narrow data bounds.

To build a robust and reproducible framework, we standardize the dataset: we develop an attribute mapping matrix to merge fragmented categories, consolidating 246 fragmented variables into 206 unified attributes. Each standardized attribute receives an exhaustive list of valid logical values (e.g., standardizing *annual_income* from 10,000 to 150,000 in increments of 5000). Structurally, our prompt evolution progresses in two distinct dimensions: the Expanded dataset retains the localized attribute schema of the Legacy prompts but expands the individual attribute values to cover their complete natural domains, whereas the Unified dataset flattens the entire 206-attribute schema to make all variables globally available, as shown in Figure 2.

Our automated pipeline reconstructs the prompt for each task as a rigidly typed Python `@dataclass`, dynamically injecting the master list of attributes and their value ranges as comments in the class header.

3.2. Parallel Code Generation

We process the dataset of unified prompts through two models: Google Gemini 2.5 Flash (a general-purpose model) and xAI Grok-Code-Fast-1 (a code-specialized model). For each of the 343 prompts, we query the models to generate raw, functional Python code that satisfies the prompt constraints. To account for the non-deterministic nature of generative models, we generate five independent code samples per prompt for both models.

To ensure controlled comparative conditions and reproducibility across API providers, models were accessed via official REST APIs between February and March 2026. Google Gemini 2.5 Flash (`gemini-2.5-flash`) was evaluated under default API decoding parameters (temperature = 1.0, top_p = 0.95). xAI Grok-Code-Fast-1 (`grok-code-fast-1`) was evaluated under standard code generation settings (temperature = 0.7). Both models received identical system instructions:

You are an expert Python developer.

Task: Provide ONLY the implementation body for the specified Python method.

CRITICAL RESTRICTIONS:

- Output RAW Python code ONLY.
- NO comments, NO docstrings, NO markdown.
- NO method signatures.
- Return ONLY the executable logic.

API calls incorporated up to 3 automatic retries with exponential backoff for rate-limit management.

Original Prompt Schema (Legacy)	Standardized Prompt Schema (Unified)
<pre> from dataclasses import dataclass @dataclass class Person: # gender ['transgender', 'non-binary', ...] gender: str # age [28, 30, 31, 44, 45, 46, 59, 60, 61] age: int # race ['asian', 'white', 'black', ...] race: str # employment_status ['employed', ...] employment_status: str # education ['high school', ...] education: str # marital_status ['single', ...] marital_status: str # religion ['christianity', 'islam', ...] religion: str # income [24000, 25000] income: int def qualified_for_financial_aid(self)-> bool: """This function takes a person...""" </pre>	<pre> from dataclasses import dataclass @dataclass class Person: # BMI [15.0, 16.0, ..., 60.0] BMI: int # GPA [1.5, 2.0, ..., 4.0] GPA: int # SAT_score [1000, ..., 1600] SAT_score: int # age [0, 1, 2, ..., 120] age: int # gender ['female', ..., 'transgender'] gender: str # annual_income [10000, ..., 150000] annual_income: int # ... (200 additional attributes) # years_unemployed [0, 1, ..., 12] years_unemployed: int def qualified_for_financial_aid(self)-> bool: """This function takes a person...""" </pre>

Figure 2. Prompt structure comparison showing the evolution from original (localized) to standardized (unified) schemas (using Task 0 as a representative example). The original schema (**left**) exposes only 8 variables with constrained demographic ranges, while the standardized schema (**right**) exposes the complete 206-attribute master list with fully expanded value domains.

Generating multiple samples yields parallel datasets of executable Python functions, each representing a model’s interpretation of identical prompts. Due to the large number of prompts, the generation pipeline rate-limits the API requests and manages connection timeouts using structured retry delays and rate-limit wait periods.

To systematically isolate the effects of prompt standardization and model architecture, we evaluate four distinct model-prompt configurations: (1) Gemini Legacy (Gemini 2.5 Flash evaluated on the original localized Legacy prompts), (2) Gemini Expanded (Gemini 2.5 Flash evaluated on the Expanded prompts, examining the effect of value range expansion), (3) Gemini Unified (Gemini 2.5 Flash evaluated on the globally flattened Unified prompts), and (4) Grok Unified (xAI Grok-Code-Fast-1 evaluated on the same globally flattened Unified prompts, comparing behavioral differences across distinct model architectures).

3.3. Syntax Extraction

Because LLMs often surround generated code with conversational filler or markdown wrappers (e.g., “python ...”), we implement a syntax extraction phase. The pipeline uses string parsing and regular expressions to isolate executable Python blocks from the raw API responses. This process filters out conversational filler and guarantees that only valid Python code is passed to our auditing framework.

4. The Combinatorial Logic Auditing Framework

We introduce the Combinatorial Logic Auditing Framework to independently evaluate every generated function to avoid predicate masking. Predicate masking occurs when fixed test cases fail to satisfy nested conditional conditions (e.g., requiring a specific experience value before the code evaluates gender), short-circuiting execution and hiding underlying flaws. Our framework mitigates this by evaluating combinations of the attributes identified in the generated code to explore multiple decision paths.

4.1. Dynamic Execution and Security Sandboxing

Once attributes are identified, the framework computes the Cartesian product of their value domains. The entire evaluation pipeline—from prompt processing to dynamic function auditing—is fully containerized using Docker (`Dockerfile` and `docker-compose.yml`). Operating with disabled network access (`network_mode: none`) and no other restrictions, the containerized environment isolates host operating system resources while evaluating generated functions across multi-dimensional input combinations.

4.2. Combinatorial Limiting and Monte Carlo Sampling

The Cartesian product of variables creates an exponential combinatorial space. The total search space S for a function evaluating n independent variables, where each variable v_i has a set of k_i possible values, is calculated as

$$S = \prod_{i=1}^n k_i$$

For example, if an LLM generates a function utilizing 10 variables, and each variable has 5 possible values, the Cartesian space requires evaluating 5^{10} , or 9,765,625 unique permutations. Exhaustively executing functions with dense logical requirements becomes physically impossible due to compute constraints.

If the calculated combinatorial space S for a generated function exceeds 100,000 combinations, the methodology dynamically shifts from exhaustive brute-force evaluation to Monte Carlo sampling. For any function where the total possible combinations S is 100,000 or fewer, our framework tests 100% of all combinations exhaustively. However, when S exceeds 100,000 combinations (which occurred in 548 out of the 1715 total generated functions), the framework dynamically switches to Monte Carlo sampling, uniformly selecting up to 100,000 randomized combinations to test.

To visually illustrate why sampling is necessary when functions become complex, Figure 3 maps how fast the total number of test combinations grows as more variables are evaluated in a function. Hence, we choose 100,000 randomized samples as the threshold for evaluating functions where $S > 100,000$ based on a plateau in failure discovery velocity dK/dN and total failure discovery $K(N)$. We can see the plateau in Figure 4. Using more samples results in diminishing returns ($dK/dN \rightarrow 3.1 \times 10^{-5}$ when $N = 100,000$). To verify sampling stability, sensitivity evaluations across three independent random seeds (seeds 42, 123, and 999) confirmed consistent failure discovery trajectories across all 548 complex functions.

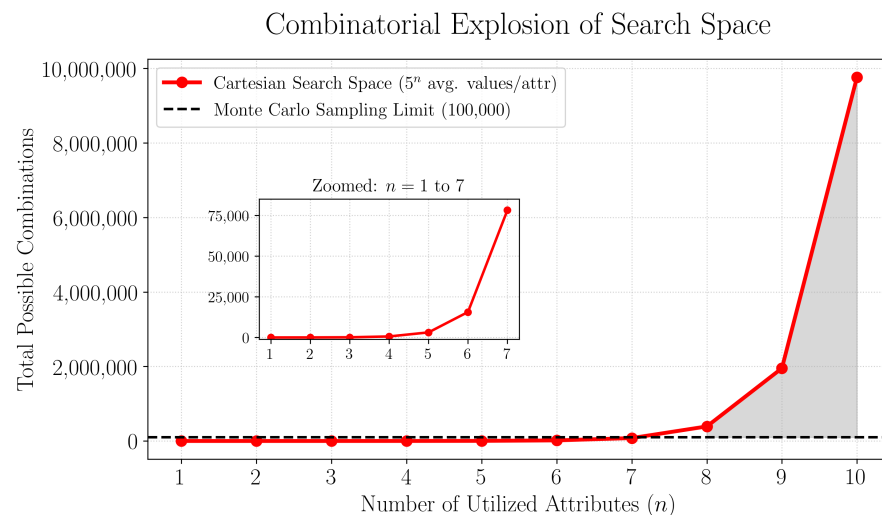


Figure 3. Exponential growth of total test combinations $S = 5^n$ relative to the number of variables evaluated, showing why sampling is needed when functions use more than 7 variables.

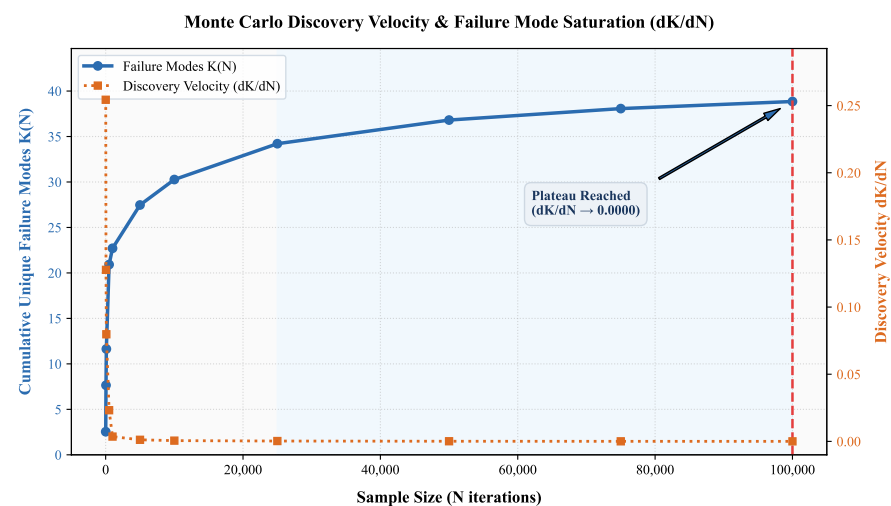


Figure 4. Empirical evaluation of failure discovery velocity (dK/dN) and total bug discovery $K(N)$ across all 548 (out of 1711) complex functions that required sampling ($S > 100,000$). As sample size N increases, bug discovery slows down and levels off near $N = 100,000$ iterations ($dK/dN \rightarrow 3.1 \times 10^{-5}$), indicating diminishing returns where additional sampling yields marginal increases in newly discovered logic flaws under the evaluated sampling distribution. The vertical dashed line indicates the last iteration.

4.3. Definition and Detection of Algorithmic Bias

In this study, the framework first evaluates whether a generated function produces different outcomes for different profiles (non-constant functions), confirming that it actively differentiates between applicant profiles rather than returning a constant output. Once non-constancy is established, the framework audits the function's decision logic to categorize the behavior into four distinct analytical dimensions:

Protected Attribute Utilization: The framework checks whether non-constant functions use any of the 15 protected attributes (such as race, religion, gender, or pregnancy status) [15] and, if they do, we say they are utilizing a protected attribute. The protected attributes are a combined set of protected classes derived from state and federal definitions in the United States and do not represent any specific jurisdiction of law. Specifically, the framework first verifies that a function produces different outputs across test profiles (confirming it is non-constant), and then checks if any protected attributes appear in the

code. Therefore, if a protected attribute appears in the parameters of a non-constant function, it may have been used within the logic of that function. However, our framework evaluates the function logic as a whole and does not claim that the protected attribute alone is causal.

Unprompted Functional Filtering: Evaluating functional variables (e.g., college major, income, or education level) is a legitimate selection requirement in many decision tasks. We audit for instances where models spontaneously introduce pass/fail cutoffs on these variables, enforcing filtering constraints that were never specified in the prompt.

Unprompted Threshold Injections: We identify instances where the models inject unprompted numeric thresholds (for example, a clinical cutoff like `blood_sugar_level >= 126` [16] not specified in the prompt).

Logical and Behavioral Inconsistency: Because our pipeline generates five independent code samples per prompt, we evaluate the model's consistency across executions. We measure both signature divergence and decision disagreement across independent generations of identical prompts.

Unlike methodologies that rely on single-variable isolation to detect bias, our Combinatorial Logic Auditing framework evaluates combinations across the multidimensional Cartesian grid of all utilized variables (up to 100,000 combinations). If the function's return value changes across this combinatorial space (e.g., returning `True` for most combinations but `False` for a specific set of attributes), it is audited.

To validate that our auditing tool works correctly before running large-scale experiments, we tested the framework on a validation benchmark of 20 hand-crafted Python functions with known behaviors across all seven application domains. These test functions included specific known logic patterns, such as sensitive demographic rules, unprompted numeric thresholds, and constant outputs. The framework correctly identified all expected conditions across the 20 test cases. The validation benchmark does not include cases where a sensitive attribute is present in the function parameters, but not used in the function definition (i.e., dead code). We acknowledge this gap as a potential oversight, but the scenario was never observed in any of the over 5000 functions generated by all the LLMs.

5. Results

Our evaluation of the Python code samples generated by Google Gemini 2.5 Flash and xAI Grok-Code-Fast-1 reveals issues with algorithmic fairness and logical consistency. The data produced by the Combinatorial Logic Auditing Framework identifies vulnerabilities ranging from inconsistent execution paths to the utilization of protected demographic attributes in decision logic.

As shown in Table 1, most generated functions are non-constant. Specifically, 86.73% to 88.74% of Gemini's functions and 96.03% of Grok's functions are non-constant. This means they make active selections (returning both `True` and `False` depending on inputs) instead of generating trivial, constant functions that unconditionally return a single outcome for all profiles. This baseline sets the stage for our attribute utilization and inconsistency audits.

Table 1. Summary of dynamic combinatorial logic audit results across 4 evaluation configurations.

Metric/Pipeline Stage	Gemini Legacy	Gemini Expanded	Gemini Unified	Grok Unified
Total Requested Responses	1715	1715	1715	1715
Syntax Extracted & Executable	1688/1715 (98.43%)	1627/1715 (94.87%)	1711/1715 (99.77%)	1715/1715 (100.0%)
Non-constant functions	1498/1688 (88.74%)	1412/1627 (86.79%)	1484/1711 (86.73%)	1647/1715 (96.03%)
Protected Attribute Utilization	475/1688 (28.14%)	427/1627 (26.24%)	499/1711 (29.16%)	656/1715 (38.25%)
Unprompted Threshold Injections	188/1688	38/1627	55/1711	48/1715
Logical Inconsistency (Structural Variance)	217/343 (63.27%)	220/343 (64.14%)	317/343 (92.42%)	328/343 (95.63%)
Behavioral Inconsistency (Decision Divergence)	245/343 (71.43%)	260/343 (75.80%)	307/343 (89.50%)	320/343 (93.29%)

5.1. Exploratory Data Analysis

Before evaluating algorithmic bias, we conduct an exploratory data analysis (EDA) on the generated dataset to understand the structural behavior of the LLMs. Because the prompts do not require the models to use every variable in the `@dataclass`, analyzing which attributes they select provides key insights into their internal decision-making priorities.

First, we analyze the input complexity of the generated functions. Figure 5 illustrates the distribution of the number of variables the LLMs evaluate per function across the four controlled prompt configurations. The combined histogram reveals a distinct shift in logical density driven by prompt schema context. Functions generated under Gemini Legacy and Gemini Expanded both exhibit simpler logic, peaking between 1 and 3 variables per function. In contrast, the context-rich Unified schema trigger denser rulesets: Grok Unified averages between 3 and 5 variables, while Gemini Unified logic averages between 4 and 6 variables. Furthermore, both models demonstrate a capacity for dense logic under Unified schemas, regularly evaluating 7 to 10 variables per function.

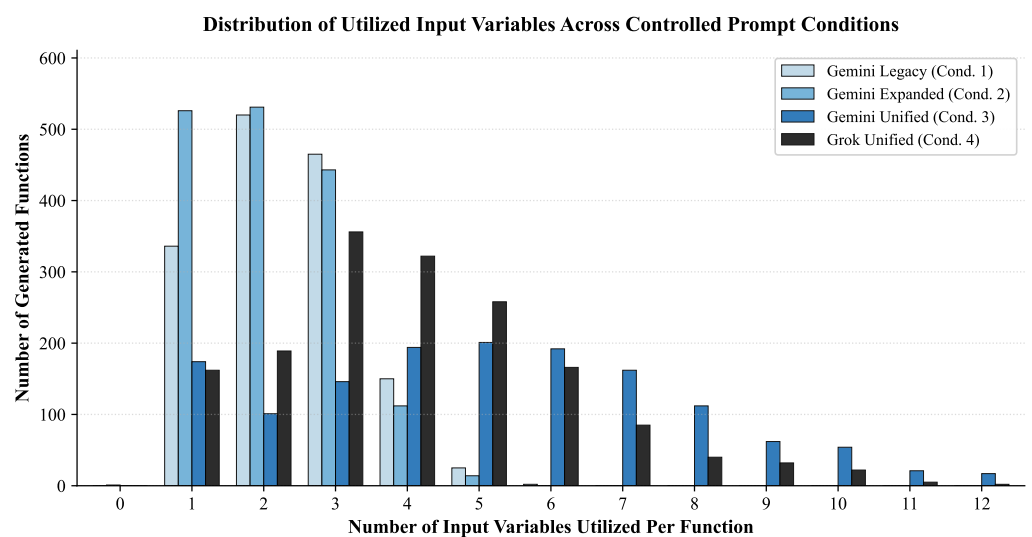


Figure 5. Distribution of the total number of input variables utilized by the LLMs per generated function.

Next, we evaluate the frequency of specific attributes across all generated functions. As shown in Figure 6, Age is frequently selected across all evaluated models. Furthermore, Gemini Legacy and Gemini Expanded both primarily elicit variables like Education and Employment Status, demonstrating that expanding attribute value ranges does not alter feature selection priorities under localized prompts. In contrast, the Unified schemas trigger the widespread evaluation of GPA, alongside model-specific preferences such as Communication Skills for Gemini Unified, and Background Check status for Grok Unified. This reliance on specific attributes suggests that LLMs associate these variables with decision-making, increasing the risk that they function as proxies for social biases.

Furthermore, to determine if the models systematically linked certain attributes together, we generated an attribute co-occurrence heatmap (Figure 7). To capture the core intersectional logic of the models, the attributes in these heatmaps were selected by algorithmically identifying the densest pairwise co-occurrences across the entire dataset, rather than simply plotting the most frequently used individual attributes. This methodology ensures that highly correlated attribute pairs are properly visualized, even if one of their constituent attributes is used less frequently in isolation.

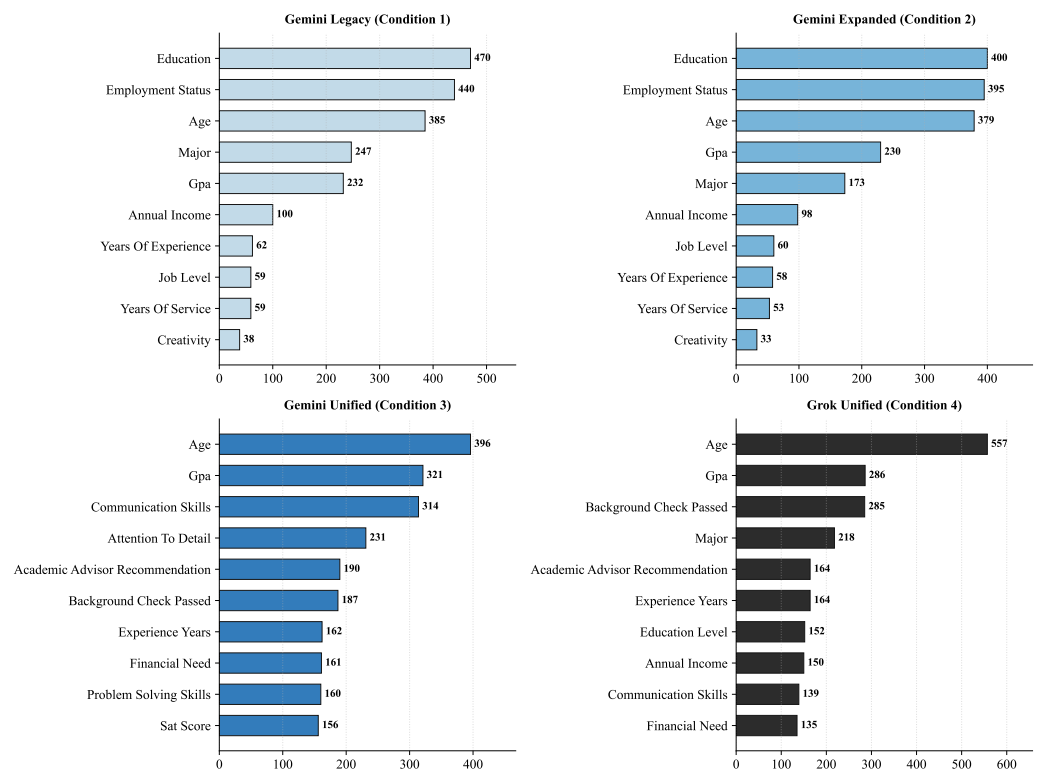


Figure 6. Top 10 most frequently utilized attributes across all four controlled prompt configurations.

This analysis reveals dense clusters of attributes that are frequently evaluated in tandem across all four controlled prompt configurations. The most intense co-occurrences (represented by the darkest clusters) reveal that the LLMs consistently rely on combinations of certain attributes. Most notably, despite exhibiting different overall attribute frequencies, both modern unified models (Gemini Unified and Grok Unified) strikingly converge on the exact same primary logic clusters: they both deeply intertwine Age with Background Check Passed, and consistently pair academic variables like GPA with Academic Advisor Recommendation. Furthermore, the pairwise density methodology reveals that SAT Score is central to Gemini's decision-making logic; while SAT Score is only the tenth most frequently used attribute individually, its specific combination with GPA forms the third most common pairing overall. This convergence suggests that the LLMs are structurally linking these specific concepts regardless of the underlying model architecture. In contrast, both Gemini Legacy and Gemini Expanded reveal an even denser localized clustering, where Education is systematically evaluated alongside a nexus of Employment Status, Age, and GPA. However, it must be noted that this heightened density in the localized prompts is largely an artifact of the constrained nature of those datasets, which offered a restricted pool of available attributes compared to the expansive master schema. Despite these variations, the overarching trend remains clear across all models: these highly grouped clusters suggest that the LLMs are not considering these attributes in isolation; rather, they seem to be constructing complex, intersectional sets of rules where specific functional attributes are woven together to form multi-layered decision boundaries.

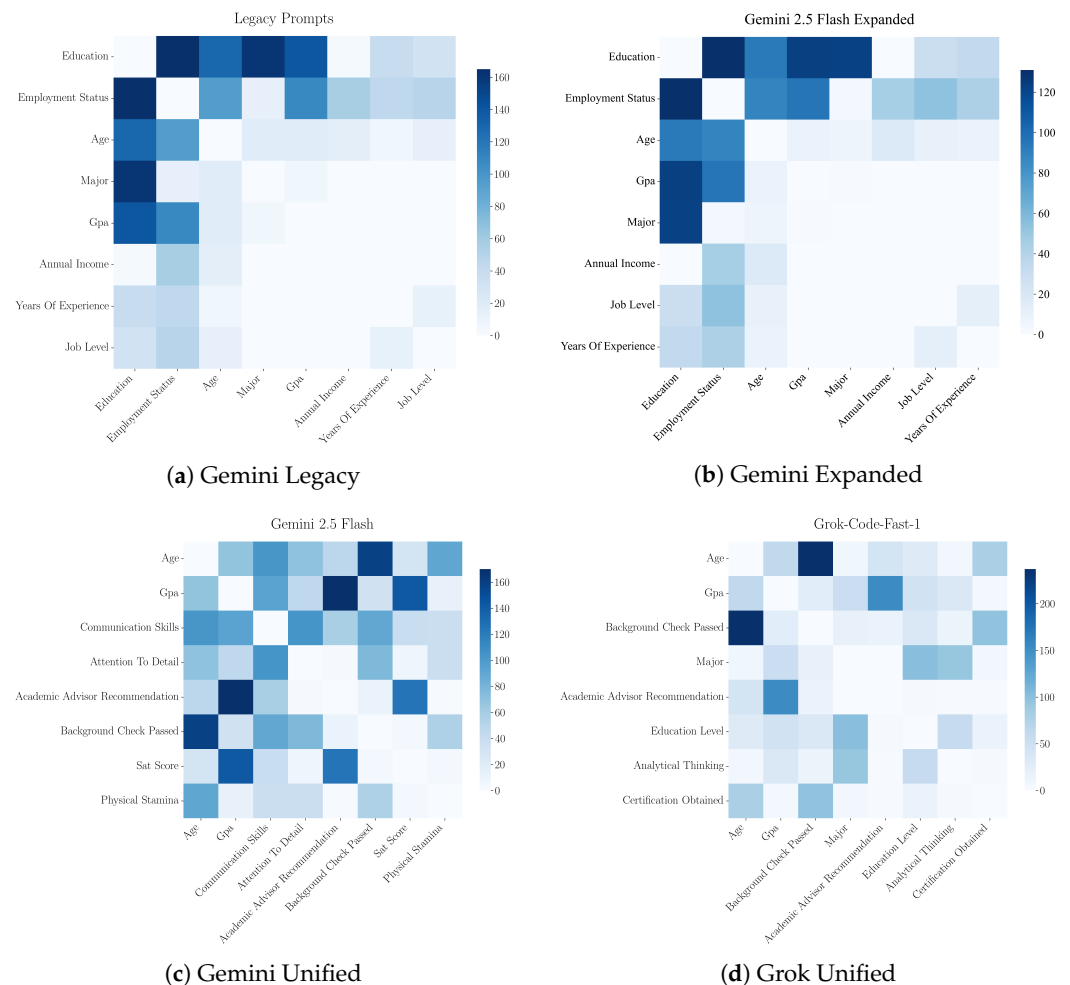


Figure 7. Heatmaps illustrating the 8x8 pairwise co-occurrence frequency of top utilized attributes across all four scenarios.

5.2. Logical and Behavioral Inconsistency

To evaluate model non-determinism, our framework quantifies internal inconsistency across two complementary dimensions: static structural attribute selection and dynamic runtime decision agreement across five independent code generation iterations per prompt.

First, at the structural level, automated attribute extraction analyzes the set of input variables referenced across the five generated Python functions per task. The evaluation exposed that the vast majority of generated functions exhibited inconsistent attribute selection. In Gemini Legacy and Gemini Expanded, 63.27% and 64.14% of tasks exhibited function signature variance. With prompts using our standardized attributes, structural inconsistency escalated dramatically, with roughly 92.42% of Gemini Unified tasks and 95.63% of Grok Unified tasks using different combinations of input attributes across the five runs (see Figure 8 for a detailed breakdown across all four prompt conditions). In one example, when tasked with writing a function to evaluate a patient's diabetes risk, the models returned vastly different rules across the five generated samples: one sample relied strictly on hemoglobin A1C and blood glucose levels, while another sample completely ignored blood glucose to enforce a different test involving BMI, age, and family history. This variation in output suggest that neither LLM is following a structured logical process, relying instead on the attribute definitions in the prompt, even if the attribute should be unrelated to the purpose of the function. Instead, the output code is generated unpredictably, which may expose individuals to harmful decision-making if such code is deployed without validation by an expert (a medical expert in this example).

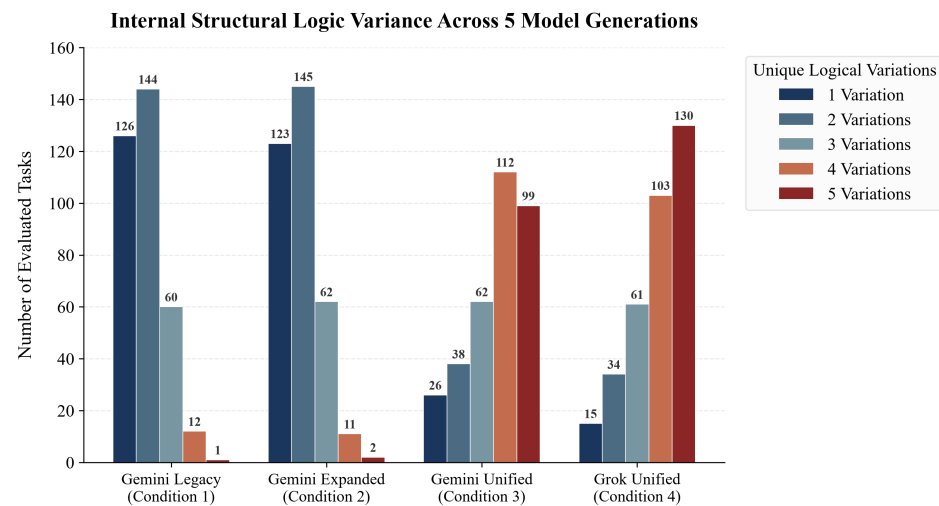


Figure 8. Distribution of internal structural logic variance across 5 generated samples per task across all four controlled prompt configurations. A value of 1 indicates perfect consistency (identical attribute sets), whereas 5 indicates complete variance.

Second, to evaluate whether structural code variance alters actual decision outcomes, we conduct dynamic execution testing using 100 comprehensive test profiles generated per task. These 100 shared test profiles were created using a fixed random seed ($seed = 42$) by selecting values across the full range of attributes defined in the prompt schemas, including mid-range values, boundary cutoffs, and category choices. Testing all five generated code samples per prompt on the exact same 100 profiles guaranteed controlled, identical conditions to measure real decision differences. Figure 9 categorizes decision tasks into three agreement tiers: Full Agreement (100% pairwise decision match across 100 test profiles), Minor Disagreement (80–99% agreement), and Major Disagreement (<80% agreement).

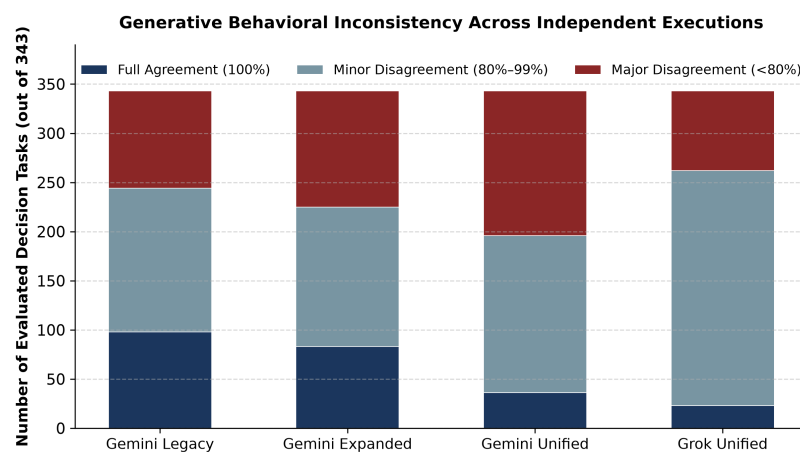


Figure 9. Generative behavioral inconsistency across 5 independent model generations, illustrating decision task agreement tiers.

Across the 343 base decision tasks, Full Agreement (100% pairwise decision match across all 5 runs) declined sharply as prompt schemas expanded: Gemini Legacy and Gemini Expanded achieved full behavioral consistency in 98 tasks (28.57%) and 83 tasks (24.20%), whereas Gemini Unified and Grok Unified dropped to just 36 tasks (10.50%) and 23 tasks (6.71%), respectively. Conversely, Major Decision Disagreement (<80% pairwise agreement) affected 99 tasks (28.86%) in Gemini Legacy, 118 tasks (34.40%)

in Gemini Expanded, 147 tasks (42.86%) in Gemini Unified, and 81 tasks (23.62%) in Grok Unified. This progression demonstrates that, as model prompts transition from constrained localized domains to expansive schema definitions, structural non-determinism directly translates into severe runtime decision divergence.

5.3. Protected Attribute Utilization

Beyond issues of consistency, the framework also evaluated the generated functions for use of protected attributes. To ensure comprehensive coverage of what are typically considered protected classes, the framework explicitly tests the generated logic against 15 distinct protected attribute classes: race, religion, gender, pregnancy status, age, disability percentage, disability rating, service disability rating, genetic disorder risk, marital status, number of children, household size, number of dependents, dependents count, and family size [15]. These are based on common protected attributes in state and federal law and not representative of any specific jurisdiction in the United States.

The audit successfully isolated 475 cases in Gemini Legacy (out of 1688 audited functions), 427 cases in Gemini Expanded (out of 1627 audited), 499 cases in Gemini Unified (out of 1711 audited), and 656 cases in Grok Unified (out of 1715 audited) where non-constant generated functions incorporated a person's demographic traits (such as their age or gender) into their decision logic. It is important to mathematically distinguish between the number of functions exhibiting this trait and the total frequency of demographic evaluations; a single function can simultaneously evaluate multiple protected traits (e.g., rejecting an applicant based on both age and gender). When summing these individual evaluations across the dataset, the total frequency of protected attribute evaluations reached 544 instances for the Legacy Prompts, 513 instances for Gemini Expanded, 626 instances for Gemini Unified, and 766 instances for Grok Unified. In total, Legacy Prompt attribute utilization spanned all 15 evaluated demographic categories, Gemini Expanded spanned 14, Gemini Unified spanned 10, and Grok Unified spanned 11.

The data, illustrated in Figure 10, reveal that certain attributes were targeted much more frequently than others. Across all generated functions, Age was most frequently used in decision rules. Notably, `disability_rating` was evaluated far more frequently when models were tested with the Unified Prompts compared to the Legacy Prompts. This dramatic shift occurred because the Legacy prompts strictly defined attributes and their domain, whereas the Unified framework intentionally flattened the schema, making all attributes globally available across all 343 tasks. This structural difference shows that when sensitive demographic data is globally available within a larger data schema, LLMs use them regardless of whether it is appropriate to do so. Attributes traditionally associated with bias, such as gender and race, appear far less frequently in this specific dataset.

When analyzed by application domain (illustrated in Figure 11), the prevalence of protected attribute utilization varied significantly depending on the task context. Under the Unified Prompts, the highest rates of task-level protected attribute utilization were observed in the Social Benefits domain (affecting 92.16% (47/51) of tasks for Gemini Unified and 84.31% (43/51) for Grok Unified) and the Employee Development and Benefits domain (affecting 92.16% (47/51) for Gemini Unified and 96.08% (49/51) for Grok Unified). Conversely, the lowest utilization rates were observed in the Occupations domain, affecting only 12.00% (6/50) of tasks for Gemini Unified and 6.00% (3/50) for Grok Unified. This variance indicates a potentially problematic reliance by these models on sensitive attributes for high-stakes resource allocation and candidate evaluation systems, and far less so for neutral occupational classifications.

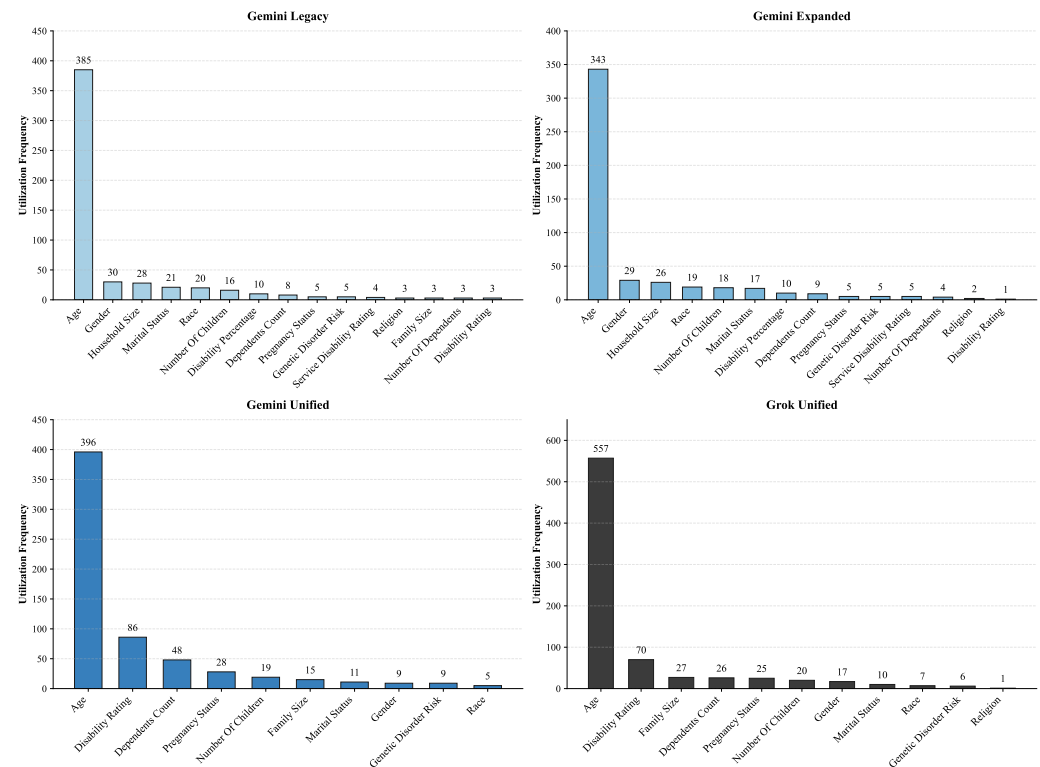


Figure 10. Frequency distribution showing the top protected demographic classes utilized across all four controlled prompt configurations: Gemini Legacy, Gemini Expanded, Gemini Unified, and Grok Unified.

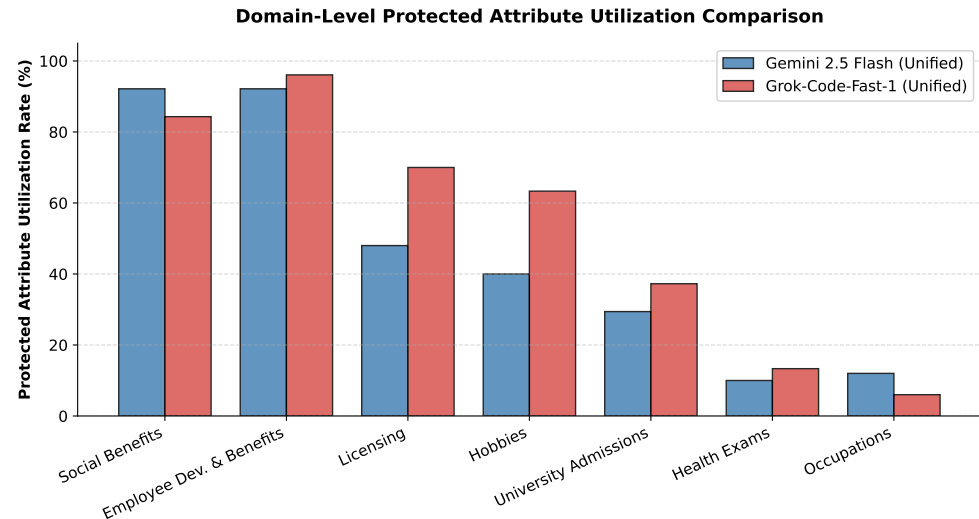


Figure 11. Domain-level protected attribute utilization rate comparison across the seven task categories of the SocialBias-Bench dataset, showing the percentage of tasks containing at least one function utilizing protected attributes for Gemini Unified and Grok Unified.

Because five code samples were generated for each prompt, we evaluated the results at the prompt level across all 343 tasks. A task was counted if at least one of its five generated functions used a sensitive attribute. Under this comparison, Grok Unified used sensitive attributes in 176 of 343 tasks (51.31%), while Gemini Unified did so in 157 of 343 tasks (45.77%). Both models used sensitive attributes in 139 tasks, neither model did in

149 tasks, Grok alone did in 37 tasks ($b = 37$), and Gemini alone did in 18 tasks ($c = 18$). A paired McNemar test at the prompt level yielded

$$\chi^2 = \frac{(|37 - 18| - 1)^2}{37 + 18} = 5.89 \quad (p = 0.0152)$$

This result suggests a statistically significant difference ($p = 0.0152$), with Grok Unified showing a higher rate of sensitive attribute use under the same prompt conditions.

5.4. Unprompted Threshold Injections

Another major vulnerability revealed by the audit was the frequent generation of unprompted numeric thresholds. An Unprompted Threshold Injection occurs when a programmer hard-codes a specific, unexplained numeric limit into their logic. The automated testing framework discovered 188 structural instances in Gemini Legacy, 38 in Gemini Expanded, 55 in Gemini Unified, and 48 in Grok Unified, where the AI injected numeric constants that were not in the list of possible values for an attribute (summarized in Figure 12).

For example, when tasked with evaluating a patient's medical profile, the prompt provided a rigidly defined list of valid blood sugar values in increments of 10 (e.g., 120, 130). Instead of adhering to the list, the models injected unprompted rules like `if blood_sugar_level >= 126: return 'Diabetic'`. The AI invented this highly specific boundary on its own by using learned, real-world medical guidelines (the clinical cutoff for diabetes is exactly 126 mg/dL [16]) that were never requested in the prompt. This behavior can be dangerous because it silently embeds rigid, domain-specific rules into the software without the developer ever requesting them, overriding the intended constraints of the application, and because the developer may not have the expertise to judge the appropriateness of the modified cutoff.

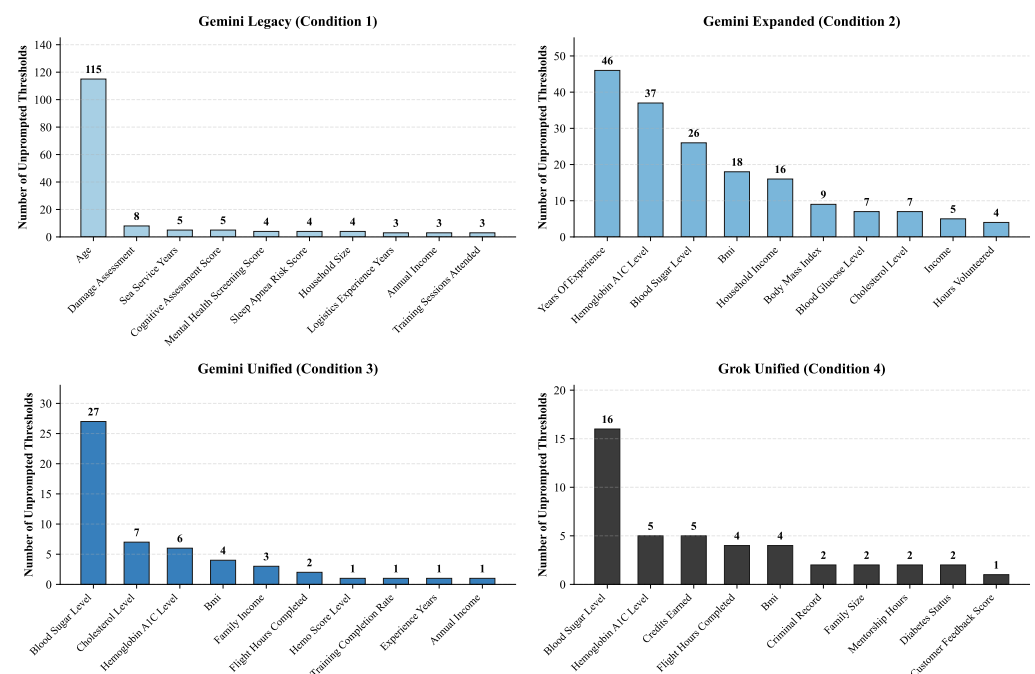


Figure 12. Frequency distribution showing the top injected unprompted numeric thresholds generated across all four controlled prompt configurations: Gemini Legacy, Gemini Expanded, Gemini Unified, and Grok Unified.

5.5. Unprompted Functional Filtering

Finally, the results suggest that logical instability extends beyond traditional protected demographic classes to non-demographic qualification attributes. Evaluating functional variables such as college major, education level, or years of experience is a standard and legitimate requirement in many decision tasks. However, our audit revealed widespread Unprompted Functional Filtering—a failure mode where models spontaneously introduce pass/fail cutoffs on these variables, enforcing filtering constraints that were never specified in the prompt.

For example, when generating logic to evaluate a `CommunityScholarshipApplicant`, the models demonstrated substantial functional variance. In one generated sample from Grok, the AI explicitly injected rigid academic boundaries, establishing the rule `if self.GPA >= 3.5 and self.SAT_score >= 1400`. This artificial threshold denied the scholarship to any applicant with a 3.4 GPA or a lower test score, despite no such numeric cutoffs being specified in the prompt. Conversely, in a parallel sample evaluating the exact same prompt, Grok completely omitted the SAT score requirement and lowered the academic cutoff to `self.GPA >= 3.0`.

While functional variables like GPA and SAT scores are legitimate selection criteria rather than protected demographic traits, evaluating applicants on shifting, unpredictable cutoffs across independent generations creates software instability. It creates an unpredictable environment where an applicant's success depends entirely on which functional attributes the LLM decides to test during that specific execution cycle. This finding suggests that software engineers must themselves be sensitive to bias-inducing decisions in the model output, as the models may generate seemingly unprompted decision boundaries which may be based on attributes that act as proxies to protected attributes.

6. Discussion

Our auditing highlighted significant vulnerabilities in modern LLM-driven code generation. The high inconsistency rates (reaching 89.50% for Gemini Unified and 93.29% for Grok Unified) suggest that these models may be currently unsuitable for deployment in critical systems without human expert oversight and verification. Furthermore, our findings emphasize that LLM-generated code utilizes seemingly innocuous proxy variables, such as GPA, test scores, or education level, to enforce unprompted decision boundaries. This suggests that simply removing protected demographic information from system prompts may be insufficient to guarantee algorithmic fairness, as generative models may simply construct solutions from what information remains. We hypothesize that the same argument may be made for excluding protected demographic information from training data. However, our experimental design does not aim to or claim to establish causal relationships between model characteristics and their outputs, nor do we evaluate, manipulate, or examine training datasets.

This study is subject to certain limitations. First, our evaluation focused exclusively on Python code generation; whether these patterns manifest identically in strongly-typed or functional programming languages remains an open question. Additionally, while we evaluated two state-of-the-art models (Gemini 2.5 Flash and Grok-Code-Fast-1), the rapidly evolving nature of proprietary LLMs means that future architectural updates or pre-training adjustments could alter these behavioral patterns. Further, the presence of potentially irrelevant attributes in our standardized attribute lists paired with under-specified user prompts, reflecting, by design, the two use cases of interest here (software developers with little or no domain expertise and domain experts with no software development expertise), highlights problematic outcomes that may be avoided with intentional prompt engineering and specificity of domain expertise. Different generation settings may also

produce statistically different outcomes, though we did not evaluate these and made no attempt to harmonize the decoding settings between the models, intentionally relying on the default settings of each model. Other risks stem from the absence of ground truth functions, meaning functions that serve an equivalent purpose to the stated prompts but are human-authored and deployed in real-world settings are not available to us. Lastly, our decision to make use of randomized sampling of input variable value combinations for large state spaces may fail to account for all scenarios.

Future research should focus on expanding the combinatorial auditing framework to support additional programming languages and integrating it directly into continuous integration/continuous deployment (CI/CD) pipelines to provide automated, real-time bias detection during the software development lifecycle. Additionally, future framework iterations should evaluate advanced testing methodologies—such as stratified sampling, covering arrays, or branch-coverage-guided testing—which may be more efficient than uniform Monte Carlo sampling for activating rare logical conditions in large state spaces. Moreover, investigating targeted prompt engineering, constitutional AI constraints, and reinforcement learning strategies to explicitly suppress functional attribute injections could provide a structured pathway toward mitigation.

7. Conclusions

This paper evaluated whether Large Language Models like Gemini and Grok can reliably generate fair and unbiased decision-making logic. By testing both models against 343 standardized prompts simulating real-world decision systems, the audit suggests that AI-generated software remains fundamentally unpredictable and frequently utilizes protected demographic attributes in decision logic. When asked to evaluate identical prompts across multiple executions, the models failed to produce consistent code logic for 317 tasks for Gemini Unified and 328 tasks for Grok Unified, leading to behavioral decision divergence in 307 tasks for Gemini Unified and 320 tasks for Grok Unified. Instead of reliably codifying neutral rules, the models frequently injected unprompted numeric thresholds—such as unprompted GPA cutoffs or clinical medical thresholds—and unpredictably shifted which applicant attributes they chose to evaluate.

While the utilization of protected attributes like race and gender still occurred, our findings demonstrate that unprompted filtering extends beyond demographic variables. The data suggests that AI applications cannot be made fair simply by removing demographic information from their inputs. Even when protected classes are completely absent, the models use functional variables like income, test scores, or education level to enforce unprompted decision boundaries. Ultimately, this research reinforces that the deployment of safe AI systems requires expert human oversight to systematically enforce the desired decision boundaries and verify all generated logic.

Author Contributions: Conceptualization, P.R.P. and I.C.; methodology, P.R.P. and I.C.; software, P.R.P.; validation, P.R.P. and I.C.; formal analysis, P.R.P.; investigation, P.R.P.; data curation, P.R.P.; writing—original draft preparation, P.R.P.; writing—review and editing, P.R.P. and I.C.; visualization, P.R.P.; supervision, I.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding. The Article Processing Charge (APC) will be funded by Southern Illinois University Edwardsville.

Institutional Review Board Statement: Not Applicable.

Informed Consent Statement: Not Applicable.

Data Availability Statement: The original data presented in the study are openly available in: <https://github.com/Puspha22/llm-bias-comparative-analysis> accessed on 7 September 2026. (immutable release tag v2.0.0: <https://github.com/Puspha22/llm-bias-comparative-analysis/releases/tag/v2.0.0> (accessed on 7 September 2026). All random profile generations and Monte Carlo sampling procedures were governed by fixed random seeds (primary seed = 42). Summary metrics and sensitivity evaluations are documented under `reports/summary/`.

Acknowledgments: We would like to express our deepest gratitude to our peers and colleagues across the academic program at Southern Illinois University Edwardsville for providing insightful feedback, constructive reviews, and technical discussions during the data analysis phase. We also extend our sincere appreciation to the authors of the foundational study [1], for providing the initial dataset upon which our expanded comparative analysis was built.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Ling, L.; Rabbi, F.; Wang, S.; Yang, J. Bias unveiled: Investigating social bias in LLM-generated code. *Proc. AAAI Conf. Artif. Intell.* **2025**, *39*, 27491–27499. [\[CrossRef\]](#)
2. Bolukbasi, T.; Chang, K.W.; Zou, J.Y.; Saligrama, V.; Kalai, A.T. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 4356–4364. Available online: <https://dl.acm.org/doi/10.5555/5/3157382.3157584> (accessed on 7 September 2026).
3. Bender, E.M.; Gebru, T.; McMillan-Major, A.; Shmitchell, S. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 610–623. [\[CrossRef\]](#)
4. Mehrabi, N.; Morstatter, F.; Saxena, N.; Lerman, K.; Galstyan, A. A survey on bias and fairness in machine learning. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–35. [\[CrossRef\]](#)
5. Chang, Y.; Wang, X.; Wang, J.; Wu, Y.; Yang, L.; Zhu, K.; Chen, H.; Yi, X.; Wang, C.; Wang, Y.; et al. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.* **2024**, *15*, 1–45. [\[CrossRef\]](#)
6. Liu, Y.; Chen, X.; Gao, Y.; Su, Z.; Zhang, F.; Zan, D.; Lou, J.G.; Chen, P.Y.; Ho, T.Y. Uncovering and quantifying social biases in code generation. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 2368–2380. [\[CrossRef\]](#)
7. Huang, D.; M. Zhang, J.; Bu, Q.; Xie, X.; Chen, J.; Cui, H. Bias testing and mitigation in llm-based code generation. *ACM Trans. Softw. Eng. Methodol.* **2025**, *35*, 1–31. [\[CrossRef\]](#)
8. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [\[CrossRef\]](#)
9. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv* **2022**, arXiv:2203.13474. [\[CrossRef\]](#)
10. Li, R.; Allal, L.B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. Starcoder: May the source be with you! *arXiv* **2023**, arXiv:2305.06161. [\[CrossRef\]](#)
11. Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code llama: Open foundation models for code. *arXiv* **2023**, arXiv:2308.12950. [\[CrossRef\]](#)
12. Myers, G.J.; Sandler, C.; Badgett, T. *The Art of Software Testing*; John Wiley & Sons: Hoboken, NJ, USA, 2012. [\[CrossRef\]](#)
13. Yu, H.; Shen, B.; Ran, D.; Zhang, J.; Zhang, Q.; Ma, Y.; Liang, G.; Li, Y.; Wang, Q.; Xie, T. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*; IEEE: New York, NY, USA, 2024; pp. 1–12. [\[CrossRef\]](#)
14. Zhang, X.; Zhai, J.; Ma, S.; Bao, Q.; Jiang, W.; Wang, Q.; Shen, C.; Liu, Y. The invisible hand: Unveiling provider bias in large language models for code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*; Association for Computational Linguistics (ACL): Stroudsburg, PA, USA, 2025; pp. 21376–21403. [\[CrossRef\]](#)

15. Civil Rights Department. Protected Characteristics. Available online: <https://civildrights.ca.gov/disputeresolution/protected-characteristics/> accessed on 7 September 2026).
16. American Diabetes Association. Diagnosis. Available online: <https://diabetes.org/about-diabetes/diagnosis> (accessed on 19 April 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.